SKJERVEN
MORRILL
MACPHERSON
FRANKLIN
& FRIEL LLP

01-18-00

A

Docket No.: M-7998 US

January 14, 2000

Box Patent Application
Assistant Commissioner for Patents
Washington, D. C. 20231

Enclosed herewith for filing is a patent application, as follows:

Inventor(s): Faisal Haq, Hari Lalgudi

Title: Implementing Access Control Lists Using A Balanced Hash Table Of Access Control List Binary Comparison Trees

X	Return Receipt Postcard
X	This Transmittal Letter (in duplicate)
26	page(s) Specification (not including claims)
12	page(s) Claims
1	page Abstract
27	Sheet(s) of Drawings
3	page(s) Declaration For Patent Application and Power of Attorney
1	page(s) Recordation Form Cover Sheet (in duplicate)
2	page(s) Assignment
2	page(s) Preliminary Amendment
74	page(s) Appendix

CLAIMS AS FILED

For	Number Filed			Number Extra		Rate			Basic Fee
								$	$690.00
Total Claims	51	-20	=	31	x	$18.00	=	$	558.00
Independent Claims	3	-3	=	0	x	$78.00	=	$	0.00
☐ Application contains one or more multiple dependent claims (total fee)								$	

Please make the following charges to Deposit Account 19-2386:

☒ Total fee for filing the patent application in the amount of $ 1,248.00

☒ The Commissioner is hereby authorized to charge any additional fees which may be required, or credit any overpayment to Deposit Account 19-2386.

EXPRESS MAIL LABEL NO:

EL252928683US

Respectfully submitted,

Dale R. Cook
Attorney for Applicants
Reg. No. 42,434

591267 v1

25 Metro Drive, Suite 700
San Jose, CA 95110
Phone 408 453-9200
Fax 408 453-7979

Austin, TX
Newport Beach, CA
San Francisco, CA

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant:	Haq, Faisal; Lalgudi, Hari K.		
Assignee:	Cisco Technology, Inc.		
Title:	IMPLEMENTING ACCESS CONTROL LISTS USING A BALANCED HASH TABLE OF ACCESS CONTROL LIST BINARY COMPARISON TREES		
Serial No.:	Unknown	Filed:	Herewith
Examiner:	Unknown	Group Art Unit:	Unknown
Atty. Docket No.:	M-7998 US		

San Jose, California
January 14, 2000

Box Patent Application
ASSISTANT COMMISSIONER FOR PATENTS
Washington, D.C. 20231

PRELIMINARY AMENDMENT

Dear Sir:

The following Amendment is submitted for entry into the application filed herewith.

AMENDMENTS

Please amend the above-referenced application as follows:

In the Specification

On page 1, at the end of line 9, please insert the following:

A portion of the disclosure (particularly the Appendix) of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright or rights whatsoever.

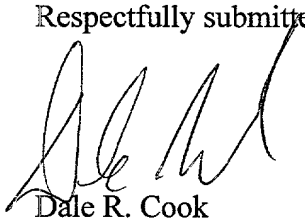
REMARKS

The application has been amended to clarify copyright issues by including the form paragraph of 37 C.F.R. 1.71 appearing in section 608.01(v) of the MPEP. No new matter has been added.

In view of the amendments and remarks set forth herein, the application is believed to be in condition for allowance and a notice to that effect is solicited. Nonetheless, should any issues remain that might be subject to resolution through a telephonic interview, the examiner is requested to telephone the undersigned.

EXPRESS MAIL NO:
EL252928683US

Respectfully submitted,



Dale R. Cook
Attorney for Applicants
Reg. No. 42,434

"Express Mail" mailing label number:

EL252928683US

IMPLEMENTING ACCESS CONTROL LISTS USING A BALANCED HASH TABLE OF ACCESS CONTROL LIST BINARY COMPARISON TREES

Faisal Haq
Hari K. Lalgudi

REFERENCE TO APPENDIX

An appendix, which will subsequently be reduced to microfiche, accompanies this application. The accompanying appendix is hereby incorporated by reference herein in its entirety.

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention is related to a method and system to be utilized in data communications involving at least one data communications network.

Description of the Related Art

Data communications is the transfer of data from one or more sources to one or more sinks that is accomplished (a) via one or more data links between the one or more sources and the one or more sinks (b) according to a protocol. A data link is the means of connecting communications facilities or equipment at one location to communications facilities or equipment at another location for the purpose of transmitting and receiving data. A protocol, in communications, computer, data processing, and control systems, is a set of formal conventions that govern the format and control the interactions between at least two communicating functional elements in order to achieve efficient and understandable communications. Examples of protocols are Asynchronous Transfer Mode (ATM) protocol, Internet Protocol (IP), and Transport Control Protocol (TCP).

A data communications network is the interconnection of three or more network stations, (each network station functioning as a data source and/or sink) over one or more data links, which allows

communication between the three or more network stations over the one or more data links. A packet-switched data communications network is a network in which data is transmitted and routed through the network in the form of packets. A packet is a sequence of bits that includes data and control signals, where typically the control signals appear in a header part -- a sequence of bits forming a first part of the packet -- and the data appear in data part -- a sequence of bits forming a second part of the packet. In packet-switched networks, data communications network stations (e.g., routers, bridges, gateways, clients, servers, etc.) may be implemented by a variety of techniques, such as software application programs running on interconnected computer systems, Application Specific Integrated Circuits (ASICs), or combinations of software and ASICs implemented within interconnected computer systems. (e.g., the Cisco Systems® Catalyst® family of switches and the Cisco 7xxx family of routers).

As noted, a data communications network is the interconnection of three or more network stations (each network station functioning as a data source and/or sink) over one or more data links. However, within the context of packet-switched networks, the convention within the art is to add to the foregoing definition the additional requirement that the defined packet-switched data communications network be under the control of a defined network administrator -- an entity (usually a person or group of persons) responsible for and having ultimate control over a defined group of network stations. Following this convention, when a first defined packet-switched network is connected to a second defined packet-switched network via at least one network station common to both the first defined and second defined networks, such a configuration is conventionally referred to as an "internetwork" -- a short hand notation for the phrase "interconnected network of networks." Note that this convention recognizes that two or more networks have been interconnected, but also recognizes that the totality of such interconnected networks itself forms a network. Thus, while the following detailed description describes devices and processes in the context of a network, such detailed description is also equally applicable to internetworks.

As networks, and networks of networks (e.g., the Internet) proliferate, increasing attention is being paid to problems involving network security (e.g., controlling which network stations can communicate with each other). For example, for a commercial lending bank having an intranet (a private network belonging to the bank)

which has one or more network stations connected to the Internet, it is common for the bank's network administrator to want to ensure that the only network stations that have access into and out of the bank's intranet are clearly defined and closely controlled. In addition, it is also common for the network administrator to restrict access between various network stations of the bank's intranet. One way that this is conventionally done is to restrict which packets can pass through each network station over which the bank's network administrator has control. In this technique, each network station examines header information of received packets in order to determine how to dispose of the packets (e.g., whether to accept, transmit, reject, or forward the received packets). By controlling, on the basis of information contained in received packet headers, which packets can pass which network stations, the network administrator is able to control access to various parts of his network on either side of each network station. Accordingly, this technique is known in the art as packet level access control.

In the packet-level access control technique, lists of rules are used by each network station to determine which received data packets to accept, transmit, forward, or reject. Since these rules control access to various portions of a network (or more or more internetworks), such rules are conventionally referred to as Access Control Rules. The complete set of rules maintained by an individual network station is conventionally known as an Access Control List (ACL).

An ACL is a set of rules for determining how a network station should dispose of various received packets. ACL rules are typically an ordered list of plain English rules which have been translated into the grammar and syntax understood by the network station where the ACL is to be implemented (e.g., expressed such that the network operating system can interpret and effect the desires expressed in the ordered list of plain English rules). For example, the plain English rule of "Permit TCP packets from any source to host with IP address equal to 194.121.68.173 and TCP port number greater than 1023" can be expressed in network station understandable grammar and syntax as "permit TCP any host 194.121.68.173 GT 1023" (expressed here for sake of example in a grammar and syntax understandable by a network server computer running Cisco Systems' IOS (Internetworking Operating System), but also expressible in other network operating system or computer operating system formats). ACLs can become quite complex and can grow to thousand upon thousands of rules.

When a data packet is received by a network station which disposes of received packets on the basis of an ACL, the packet's header information must be compared against those ACL rules which utilize the information contained within the received packet's header in order to make access control decisions. In addition, such comparisons should be done in the sequential order in which the rules appear in the ACL, since the order in which the rules are arranged in an ACL typically encodes important control information (e.g., if an ACL has a first-in-sequence rule that states "permit packets from source address Memphis to destination address San Francisco," and a second-in-sequence rule that states "deny packets from source address Memphis to any destination address," if the order of evaluation is reversed, the packet from Memphis to San Francisco will never get through the network station).

For an ACL with a relatively small number of rules, comparing a packet's header against the ACL rules causes very little network traffic delay. However, as the number of rules in an ACL grows, the delay associated with comparing packet headers against the ACL rules can be very computationally intensive, and can result in significant network traffic delay above and beyond that associated with smaller ACLs.

While it may seem reasonable that ACLs with a relatively large number of rules will result in significant network delay above and beyond that associated with ACLs with a relatively smaller number of rules, those skilled in the art will recognize that network administrators prefer that adding rules to ACLs maintained by network stations not result in noticeable performance degradation of those network stations.

Techniques exist within the art which provide network stations with the ability to maintain relatively larger ACLs without noticeable degradation of performance over relatively smaller ACLs, but those techniques are not practicable for many situations. For example, some of the most successful techniques have involved a faster and more compact method of converting the ACL rule elements into entries in a content-addressable memory (CAM), but such techniques tend to add overhead to the systems. Furthermore, insofar as that the CAM-based techniques involve a radical design departure from older-generation systems, the newer CAM-based network stations are generally not backwards-compatible with existing network stations.

In many situations, network administrators are dissatisfied with the ACL-related performance of their network stations, but such network administrators have either concluded that the problems are not severe enough to warrant investing in the newer CAM-based network stations, or such network administrators cannot afford the newer CAM-based network stations. In addition, many network station vendors have invested significant research and development funds into their current-generation network stations, and would prefer to extend the life of such current-generation network stations before adopting the radical redesign needed to move to the newer-generation CAM-based network stations. One way of extending the life of such current-generation network stations would be to improve the ability of such network stations to handle relatively long ACLs.

It is therefore apparent that a need exists for a method and system which will provide improved ACL performance for network stations in a relatively cost-effective manner (e.g., in a manner not requiring the use of expensive CAM-based technology). In addition, a need further exists for a method and technique which is substantially backwards-compatible with existing ACL systems, which will thus allow vendors to retro-fit network stations already purchased by customers, and also allow vendors to further extend the life of their current-generation network stations.

The foregoing general discussion of the related art can be supplemented by reference to the following texts, all of which are hereby incorporated by reference in their entireties: Merilee Ford, et. al., Internetworking Technologies Handbook, Cisco Press 1997; Karanjit S. Siyan, Inside TCP/IP, 3d ed., New Riders Publishing 1997; Internet Firewalls and Network Security, 3d ed., New Riders Publishing 1995; D. Brent Chapman and Elizabeth D. Zwicky, Building Internet Firewalls, O'Reilly & Associates, 1995; Network Protocols Configuration Guide, Cisco IOS® Release 12.0, Cisco Press, 1998; and Network Protocols Command Reference, IOS Release 12.0, Cisco Press, 1998.

Cisco Systems, Cisco IOS, and Catalyst are registered trademarks of Cisco Systems, Inc. of San Jose, California.

SUMMARY OF THE INVENTION

The inventors named herein have devised a method and system which provide improved ACL performance for network stations (e.g.,

routers, bridges, switches, network clients, and network servers) in a relatively cost-effective manner (e.g., in a manner not requiring the use of expensive CAM-based technology). The method and system devised are substantially backwards-compatible with existing-generation network stations, and will allow vendors to retro-fit network stations already purchased by customers, and further extend the life of current generation systems sold by such vendors.

The devised method and system provide for implementing Access Control Lists (ACLs) using a Balanced Hash Table of ACL Binary Comparison Trees (ABCTs), where the Balanced Hash Table of ABCTs encodes the replaced ACL. In one embodiment, the method includes but is not limited to receiving at least one packet, and disposing of the received at least one packet in response to a walk of a Balanced Hash Table of ABCTs, where the Balanced Hash Table of ABCTs encodes an Access Control List. In another embodiment, the method further includes converting the Access Control List to the Balanced Hash Table of ABCTs, the Balanced Hash Table of ABCTs encoding the Access Control List. In one embodiment, the system includes means for receiving at least one packet, and means for disposing of the received at least one packet in response to a walk of a Balanced Hash Table of ABCTs, where the Balanced Hash Table of ABCTs encodes an Access Control List. In another embodiment, the system further includes means for converting the Access Control List to the Balanced Hash Table of ABCTs, where the Balanced Hash Table of ABCTs encodes the Access Control List.

The foregoing is a summary and thus contains, by necessity, simplifications, generalizations, and omissions of detail; consequently, those skilled in the art will appreciate that the summary is illustrative only and is not intended to be in any way limiting. Other aspects, inventive features, and advantages of the present invention, as defined solely by the claims, will become apparent in the non-limiting detailed description set forth below.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings.

Figure 1A depicts a high level logic flowchart depicting a process by which a Balanced Hash Table of ACL Binary Comparison Trees

is constructed and thereafter utilized within a Per-Packet Processing Engine of a network station (exemplary of any one of many network stations well-known in the art, and which may include, among other things, all or part of the following: a processor, router, bridge, switch, gateway, network server, or network client).

Figure 1B depicts a pictorial representation of an example of network station 150

Figure 2 illustrates a high-level logic flowchart depicting the ACL Hash-Table-Balancing Bit Selection Vector Production Process.

Figures 3A and 3B show a high-level logic flowchart illustrating the "heuristic balancing process" referred to in method step 208.

Figure 4 depicts a high-level logic flowchart illustrating the ACL-to-Balanced Hash Table of ACL Binary Comparison Tree Conversion Process.

Figure 5 illustrates a process by which a binary comparison tree may be constructed for a Rule Under Consideration, such as was referenced in relation to method step 410.

Figure 6 illustrates a process by which the binary comparison tree constructed for the Rule Under Consideration may be added to a hash table.

Figures 7A-7D12 show an example of the creation of a Hash-Table-Balancing Bit Selection Vector, and the subsequent creation of a Balanced Hash Table of ABCTs.

The use of the same reference symbols in different drawings indicates similar or identical items.

DETAILED DESCRIPTION

With reference to the figures and in particular with reference now to Figure 1A, depicted is a high-level logic flowchart depicting a process by which a Balanced Hash Table of ACL Binary Comparison Trees is constructed and thereafter utilized by a per-packet processing engine within network station 150 (where network station 150 is exemplary of any one of many network stations well-known in the art, and which may include, among other things, all or part of the following: a processor, router, bridge, switch, gateway, network

server, or network client). Method step 100 illustrates the start of the process. Method step 102 shows obtaining a complete, ordered, ACL utilized within network station 150. Those having ordinary skill in the art will recognize that such an ACL is typically entered by a human network administrator via a graphical user interface of an application program resident upon a network server computer, such network server computer typically having software and/or hardware sufficient to allow it to function as a network router, gateway, and/or bridge.

Method step 104 shows an ACL Hash-Table-Balancing Bit Selection Vector Production Process (discussed in more detail via flowcharts and a specific example, below) receiving the complete, ordered set of ACL Rules. Method step 106 depicts that the ACL Hash-Table-Balancing Bit Selection Vector Production Process outputs and passes to the next method step definitions of certain bit positions within one or more packet header fields utilized by the rules in the ACL, such definitions contained as pointers within what is hereinafter referred to as a "Hash-Table-Balancing Bit Selection Vector" which will be utilized (1) to construct a hash table having a relatively balanced set of entries of ACL binary comparison trees(as used herein "balanced" means that the trees are distributed roughly evenly both in depth and across the entries of the entire hash table), where such constructed hash table will encode the ACL (a process explained in more detail via flowcharts and a specific example, below) and, (2) to thereafter construct hash table index values used to "key into" the constructed Balanced Hash Table of ACL Binary Comparison Trees once the Balanced Hash Table of ACL Binary Comparison Trees has been successfully deployed in per-packet processing engine 152.

Method step 108 shows that an ACL-to-Balanced Hash Table of ACL Binary Comparison Tree Conversion Process (discussed in more detail below by way of a flowchart and a specific example), receives the Hash-Table-Balancing Bit Selection Vector specified by the ACL Hash-Table-Balancing Bit Selection Vector Process and also receives the complete, ordered, ACL. Thereafter, method step 108 depicts that the ACL-to-Balanced Hash Table of ACL Binary Comparison Tree Conversion Process, utilizing the Hash-Table-Balancing Bit Selection Vector and the complete, ordered, ACL, creates and outputs what is referred to herein as a Balanced Hash Table of ACL Binary Comparison Trees which actually encodes the ACL, such Balanced Table of ACL Binary Comparison Trees being hereinafter referred to as a Balanced Hash

Table of ABCTs (the production of which is discussed in more detail below by way of a flowchart and a specific example).

Method step 110 depicts that the ACL-to-Balanced Hash Table of ACL Binary Comparison Tree Conversion Process outputs and passes to the next method step a Balanced Hash Table of ABCTs, which method step 152 shows is thereafter accepted by a per-packet processing engine resident within and utilized by a network station 150 to effect the mandates encoded in the ACL rules.

Event 112 illustrates that subsequent to acceptance of the Balanced Hash Table of ABCTs by per-packet processing engine 150, network station 150 receives packets, and the per-packet processing engine utilizes the Hash-Table-Balancing Bit Selection Vector to create from the header of the received packets a hash table index value which is used to "key into" the Hashed Table of Balanced ACL Trees, and thereafter walk the Balanced Hash Table of ABCTs to determine the appropriate disposition of the received packet as dictated by the complete, ordered, ACL. Event 114 depicts that the walk of the Balanced Hash Table of ABCTs results in disposition of the received packets, referenced in event 112, in the manner indicated by the ACL.

With reference now to Figure 1B depicted is a pictorial representation of an example of network station 150. Depicted is network station 150. Shown present and associated with network station 150 are computer system unit 122 respectively coupled to video display device 124, keyboard 126, mouse 127, and router 128. Depicted is that router 128 is connected with communication lines 130 whereby one or more data packets may enter and exit network station 150. Network station 150 may be implemented utilizing any suitable computer system (e.g., workstations, minicomputers, mainframe computers, or network-application specific computers) and router. Those skilled in the art will recognize that various implementations of network station 150 can have many different components, such as multiprocessors, random-access memory, read-only memory, various bus types, disk and tape drives, graphical user interfaces, etc. In addition, network station 150 is merely exemplary and it is to be remembered that network stations referred to herein may be implemented by a variety of techniques, including but not limited to software application programs running on interconnected computer systems, Application Specific Integrated Circuits (ASICs), or combinations of software and ASICs implemented within interconnected

computer systems. (e.g., the Cisco Systems® Catalyst® family of switches and the Cisco 7xxx family of routers).

Referring now to Figure 2, illustrated is a high-level logic flowchart depicting the ACL Hash-Table-Balancing Bit Selection Vector Production Process. Method step 200 shows the start of the process. Method step 202 depicts obtaining the complete, ordered, ACL (previously referenced in relation to method step 102) utilized within network station 150. Method step 203 illustrates identifying the packet header fields which are utilized by the ACL Rules in order to make access control decisions. Thereafter, method step 204 depicts scanning each packet header utilized by each ACL rule in the complete, ordered, ACL, and constructing an exemplar aggregate bit string, where the exemplar aggregate bit string has one field corresponding to each packet header field utilized by any ACL Rule in the ACL and such that no fields are duplicated within the exemplar bit string (that is, even if two different ACL rules utilize the same packet header field, only one instance of the packet header field will appear in the exemplar, even though two rules use that field).

Thereafter, method step 206 depicts constructing, by using the packet identification criteria (i.e., fields) utilized by each ACL rule in the complete, ordered, ACL referenced in method step 202, a bit string having the same fields as the constructed exemplar aggregate bit string referenced in method step 204, where each constructed bit string for each rule has as the contents of its (the constructed bit string's) fields the contents of the fields utilized for each such rule, and "don't care" entries for fields not utilized by each such ACL Rule.

Method step 208 shows obtaining, or alternatively specifying, the number of bits (or "bit length") to be utilized in an index of a yet-to-be constructed Balanced Hash Table of ABCTs. Thereafter, method step 210 depicts utilizing a "heuristic balancing" process (discussed in more detail below by way of a flowchart and a specific example) to construct a Hash-Table-Balancing Bit Selection Vector which has pointers (the number of pointers selected is equal in number to the specified bit length of the hash table index) to those bit positions utilized by the ACL rules, where the "heuristic balancing" process selects those bit positions which both (1) are utilized with relative frequency by the ACL Rules *and* (2) have entries within the selected bit positions that are roughly equally distributed amongst logical "1"s (ones) and logical "0"s (zeroes);

With reference now to Figures 3A and 3B, shown is a high-level logic flowchart illustrating the "heuristic balancing process" referred to in method step 208. Method step 300 depicts the start of the process. Method step 302 illustrates obtaining the bit strings, referenced in method step 206, constructed for each ACL rule.

Method step 304 shows aligning the bit positions of each obtained constructed bit string referenced in method step 302; that is, each constructed bit string is aligned such that each field in the constructed bit string matches with the appropriate field in every other constructed bit string, and thus each bit position within each constructed bit string is also so-aligned -- the foregoing is possible because each bit string for each rule was constructed utilizing the exemplar bit string referenced in method step 204 of Figure 2.

Method step 306 depicts that the leftmost aligned bit position within the aligned bit strings is defined to be the "current aligned bit position." Thereafter, method step 308 illustrates that, for the current aligned bit position (1) a count is made of the number of "1"s (logical ones) appearing in that current aligned bit position amongst all the bit strings constructed from the ACL Rules in the ACL, (2) a count is made of the number of "0"s (logical zeroes) appearing in that current aligned bit position amongst all the bit strings constructed from the ACL Rules in the ACL, and (3) a count is made of the number of "X"s (logical "don't care" bits) appearing in that current aligned bit position amongst all the bit strings constructed from the ACL Rules in the ACL.

Method step 310 shows that once a count has been made of the "1"s, "0"s, and "X"s appearing in the current aligned bit position amongst all the bit strings constructed from the ACL Rules in the ACL, the count of the "1"s is summed with the count of the "X"s to obtain a "Total '1's + 'X's Count" for the current aligned bit position; also shown is that the count of the "0"s is summed with the count of the "X"s to obtain a "Total '0's + 'X's Count" for the current aligned bit position.

Method step 312 depicts an inquiry as to whether the current aligned bit position is the last position in the aligned constructed bit strings. Method step 314 shows that if the inquiry of method step 312 yields a determination that the current aligned bit position is NOT the last position in the constructed bit strings, the current aligned bit position is redefined as the next-rightwards bit position

of the aligned constructed bit strings. Thereafter, the process proceeds to method step 308 and continues from that point.

Method step 316 depicts that if the inquiry of method step 312 yields a determination that the current aligned bit position is the last position in the constructed bit strings, a "Larger Total Count" row (in one embodiment, 1 x N matrix, where N is equal to the total number of bit positions in the exemplar bit string referenced in method step 204 above) having one column corresponding to each bit position in one of the constructed bit strings (i.e., a column for each bit position in the exemplar bit string reference in method step 204) is created.

Method step 318 shows that each column entry of the "Larger Total Count" row is filled with the LARGER of either the "Total '1's + 'X's Count" or the "Total '0's + 'X's Count" for the bit positions of the constructed bit strings corresponding to each such column entry of the Larger Total Count row.

Method step 320 illustrates that a "Smaller Total Count" row (in one embodiment, 1 x N matrix, where N is equal to the total of bit positions in the exemplar bit string referenced in method step 204 above) having one column corresponding to each bit position in one of the constructed bit strings (i.e., a column for each bit position in the exemplar bit string reference in method step 204) is created.

Method step 322 shows that each column entry of the "Smaller Total Count" row is filled with the SMALLER of either the "Total '1's + 'X's Count" or the "Total '0's + 'X's Count" for the bit positions of the constructed bit strings corresponding to each such column entry of the Smaller Total Count row.

Method step 323 shows that the number of Unspecified Pointers of Hash-Table-Balancing Bit Selection Vector is set equal to the bit length of the hash table index specified in method step 208 of Figure 2.

Method step 324 depicts that the one or more columns of the Larger Total Count row referenced in method step 318 having the minimum value within the Larger Total Count row are selected and designated as potential, "P," candidate columns which will be used to construct the pointers of the Hash-Table-Balancing Bit Selection Vector, which means that the bit positions (of the constructed bit

strings/exemplar bit string) corresponding to the selected columns are potential candidates for use as the Hash Table Index.

Method step 326 illustrates an inquiry as to whether the number of potential, "P," candidate columns selected in method step 324 is greater than the number of Unspecified Pointers of Hash-Table-Balancing Bit Selection Vector. In the event that the inquiry illustrated in method step 326 yields a determination that the number of potential, "P," candidate columns selected in method step 324 is greater than the number of Unspecified Pointers of Bit Position Vector, the process proceeds to method step 328 which shows that an attempt is made to refine the selection of the potential, "P," candidate columns referenced in method step 324 by examining the one or more columns of the Smaller Total Count row, such examined Smaller Total Count row columns being those corresponding to the columns currently designated as potential, "P," candidate columns within the Larger Total Count row; further shown is that those examined columns within the Smaller Total Count row with the minimum, or smallest, entries are redesignated as potential, "R," candidate columns. Thereafter, method step 330 shows that an inquiry is made as to whether the total number of redesignated potential, "R," candidate columns is greater than or equal to the number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector.

In the event that the inquiry of method step 330 yields a determination that the number of redesignated potential, "R," candidate columns are greater than or equal to the number of the Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector, the process proceeds to method step 332 which depicts that a number, equal to the number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector, of potential "R" candidate columns are designated as actual "K" Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns, which means that the bit positions corresponding to those "K" candidate columns are those bit positions which will be pointed at by pointers of the Hash-Table-Balancing Bit Selection Vector. Thereafter, insofar as that the Hash-Table-Balancing Bit Selection Vector has now been completely specified, the process proceeds to method step 333 and stops.

In the event that the inquiry of method step 330 yields a determination that the number of redesignated potential, "R," candidate columns are less than the number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector, the process proceeds

to method step 334 which depicts that the potential "P" candidate columns are designated as actual "K" Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns, which means that the bit positions corresponding to those "K" Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns are those bit positions which will be pointed at by pointers of the Hash-Table-Balancing Bit Selection Vector. Subsequently, method step 336 illustrates the number of actual "K' Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns referenced in method step 332 is subtracted from the number of Unspecified Pointers of Hash-Table-Balancing Bit Selection Vector.

Method step 338 shows the inquiry as to whether the number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector is greater than zero. If the inquiry depicted in method step 338 results in a determination that the number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector is NOT greater than zero, the process proceeds to method step 340 and stops. If the inquiry depicted in method step 338 results in a determination that the number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector is greater than zero, the process proceeds to method step 342 which depicts that the columns of "Larger Total Count" Row and the columns "Smaller Total Count" Row which have previously been designated as "K" Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns in any method step, at any iteration through the process depicted in Figures 3A and 3B, are marked as "no longer selectable" (this ensures that the columns/bit positions subsequently designated as "K" Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns will be bit positions that have not yet been so designated). Thereafter, with Larger Total Count row and Smaller Total Count row so adjusted (that is, with the Larger Total Count and Smaller Total Count row columns which have ever been designated as "K" Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns being marked as no longer selectable, or under consideration), the process proceeds to method step 324 and continues from that point.

Notice that the process will loop until the number of candidates "K" necessary to completely specify the Pointers of Hash-Table-Balancing Bit Selection Vector have been designated.

In the event that the inquiry illustrated in method step 326 yields a determination that the number of columns selected in method

step 324 is not greater than the number of Unspecified Pointers of Bit Position Vector, the process proceeds to method step 334, which depicts that the potential, "R," candidate columns are designated as actual "K" Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns, which means that the bit positions corresponding to those "K" Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns are those bit positions which will be pointed at by pointers of the Hash-Table-Balancing Bit Selection Vector. Thereafter, the process proceeds to method step 336 and continues from that point.

Referring now to Figure 4, depicted is a high-level logic flowchart illustrating the ACL-to-Balanced Hash Table of ACL Binary Comparison Tree Conversion Process. Method step 400 shows the start of the process. Method step 402 depicts that the complete, ordered, ACL referenced within method step 102 is obtained. Method step 404 illustrates the creation of a hash table having a number of entries corresponding to the bit length specified for the hash table index (i.e., that specified in method step 208); for example, a 2 bit length index would have a hash table with 4 entries, a 3 bit length index would have a hash table with 8 entries, a 4 bit index would have a hash table with 16 entries, etc. Method step 406 shows that a "Rule Under Consideration" is set to be the first rule in the ACL. Method step 408 shows that the bit positions, previously designated as "K" Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns (i.e., the bit positions pointed at by the pointers of the Hash-Table-Balancing Bit Selection Vector), in the fields utilized by the Rule Under Consideration are examined. Method step 409 depicts that that the bit entries in those examined bit positions (i.e., those bit positions pointed at by the pointers of the Hash-Table-Balancing Bit Selection Vector) are utilized to construct an hash table index value, used to "key into" the created hash table. Those skilled in the art will recognize that there are many ways such could be done, but one way would be to utilize the hash table index as part of the memory address of the table.

Method step 410 shows that once the appropriate row entry of the hash table has been keyed to using the hash table index value for the Rule Under Consideration, the fields utilized by the Rule Under Consideration are examined in a left-to-right fashion, and a binary comparison tree is constructed for the Rule Under Consideration (construction of a binary comparison tree is illustrated via a flowcharts and a specific example, below). Thereafter, method step

412 (discussed in more detail via flowcharts and a specific example, below) shows that the constructed binary comparison tree for the Rule Under Consideration is appended to the hash table entry associated with the hash table index equal to the hash table index value constructed for the Rule Under Consideration. With respect to the constructed hash table index value, if one of the bit positions of the constructed hash table index value contains an "X" value, then the tree for the Rule Under Consideration will be appended in the hash table both where such bit position index equals "1" and "0"; for example, if an index turned out to be 11X, the binary comparison tree for the Rule Under Consideration would be appended to the hash table index values 111 *and* 110.

Method step 414 depicts the inquiry as to whether the end of the complete, ordered, ACL referenced within method step 102 has been reached. If the inquiry depicted in method step 414 yields a determination that the end of the complete, ordered, ACL has NOT been reached, method step 416 shows that the Rule Under Consideration is set to be the next ACL rule in sequence within the complete, ordered, ACL. Thereafter, the process proceeds to method step 406 and continues from that point.

If the inquiry depicted in method step 414 yields a determination that the end of the complete, ordered, ACL has been reached, method step 418 depicts that the resultant hash table with the appended binary trees is designated as a Balanced Hash Table of ABCTs. Thereafter, the process proceeds to method step 420 and stops.

With reference now to Figure 5, illustrated is a process by which a binary comparison tree may be constructed for a Rule Under Consideration, such as was referenced in relation to method step 410. The following method steps refer to inserting compare nodes in a binary comparison tree. It is to be understood that when a node is so inserted, the insertion is such that the miss branch of the inserted node will point to "DEFAULT DENY" and the match branch of the inserted node will point to a place-holder for the next node to be inserted, unless the insertion is related to the insertion of a "stop" node (a "stop" node does not have a miss or match branch but instead contains the ultimate dispensation of the rule, such as "PERMIT," "DENY," "FORWARD," etc.).

Method step 500 shows the start of the process. Method step 502 depicts the inquiry as to whether the Rule Under Consideration

makes a decision based upon the protocol field of received packet headers. In the event that the inquiry depicted in relation to method step 502 yields a determination that a decision is NOT made based upon the protocol field of received packet headers, the process proceeds to method step 506 and continues from that point. In the event that the inquiry depicted in relation to method step 502 yields a determination that a decision is made based upon the protocol field of received packet headers, the process proceeds to method step 504 wherein is depicted the operation of inserting a protocol compare node, having separate miss and match branches consistent with the protocol field of the Rule Under Consideration, into a binary comparison tree (notice that this is the first insertion of a compare node into the binary compare tree). Thereafter, the process proceeds to method step 506 wherein is depicted the inquiry as to whether the Rule Under Consideration makes a decision based upon the source address field of received packet headers.

In the event that the inquiry depicted in relation to method step 506 yields a determination that a decision is NOT made based upon the source field of received packet headers, the process proceeds to method step 510 and continues from that point. In event that the inquiry depicted in method step 506 yields a determination that the Rule Under Consideration does make a decision based upon the source address field of received packet headers, the process proceeds to method step 508 wherein is depicted that a source address compare node, having separate miss and match branches consistent with the source address field of the Rule Under Consideration, is appended to the preceding compare node. Thereafter, the process proceeds to method step 510 wherein is depicted the inquiry as to whether the Rule Under Consideration makes decisions based upon the source port field of received packet headers.

In the event that the inquiry depicted in relation to method step 510 yields a determination that a decision is NOT made based upon the source port field of received packet headers, the process proceeds to method step 514 and continues from that point. In event that the inquiry depicted in method step 510 yields a determination that the Rule Under Consideration does make a decision based upon the source port field of received packet headers, the process proceeds to method step 512 wherein is depicted that a source port compare node, having separate miss and match branches consistent with the source port field of the Rule Under Consideration, is appended to the preceding compare node. Thereafter, the process proceeds to method

step 514 wherein is depicted the inquiry as to whether the Rule Under Consideration makes decisions based upon the destination address field of received packet headers. In the event that the inquiry depicted in relation to method step 514 yields a determination that a decision is NOT made based upon the destination address field of received packet headers, the process proceeds to method step 518 and proceeds from that point. In event that the inquiry depicted in method step 514 yields a determination that the Rule Under Consideration does make a decision based upon the destination address field of received packet headers, the process proceeds to method step 516 wherein is depicted that a destination address compare node, having separate miss and match branches consistent with the destination address field of the Rule Under Consideration, is appended to the preceding compare node. Thereafter, the process proceeds to method step 518 wherein is depicted the inquiry as to whether the Rule Under Consideration makes decisions based upon the destination port field of received packet headers. In the event that the inquiry depicted in relation to method step 518 yields a determination that a decision is NOT made based upon the destination port field of received packet headers, the process proceeds to method step 519, which shows the insertion of a stop node having miss and match branches consistent with final dispensation of Rule Under Consideration. Thereafter, the process proceeds to method step 522 and stops. In event that the inquiry depicted in method step 518 yields a determination that the Rule Under Consideration does make a decision based upon the destination port field of received packet headers, the process proceeds to method step 520 wherein is depicted that a destination port compare node, having separate miss and match branches consistent with the destination port field of the Rule Under Consideration, is appended to the preceding compare node. Thereafter, the process proceeds to method step 521, which shows the insertion of a stop node having miss and match branches consistent with final dispensation of Rule Under Consideration. Thereafter, the process proceeds to method step 522 and stops.

For sake of simplicity the process described in relation to Figure 5 makes reference only to source address, source port, destination address, destination port, and protocol identification fields. However, it is to be understood and will be appreciated by those having skill in the art that many other such fields exist (e.g., IP Quality of Service, TCP Flag fields) which can be utilized to construct binary comparison trees in a fashion substantially analogous to that demonstrated in Figure 5. The present discussion

refers to logical nodes having miss and match branches; in one embodiment such nodes having miss and match branches are implemented as an instruction word to a packet processor containing a compare opcode and operand. Accordingly, Figure 5, as all figures herein, is intended to be exemplary and not limiting.

Referring now to Figure 6, illustrated is the process, referenced in method step 412, by which the binary comparison tree constructed for the Rule Under Consideration may be added to a hash table. Method step 600 shows the start of the process. Method step 602 depicts the inquiry as to whether there is a pre-existing binary comparison tree at the hash table index equal to the hash table index value created for the ACL Rule Under Consideration. If the inquiry of method step 602 yields a determination that there is NOT a binary comparison tree at the hash table index equal to the hash-table-index value index created for the ACL Rule Under Consideration, method step 604 illustrates that the binary comparison tree is appended in its entirety at the hash table entry associated with the hash table index, with the leftmost node of the binary comparison tree serving as root of the tree. Thereafter, the process proceeds to method step 606 and stops.

If the inquiry depicted in method step 602 yields a determination that there is a binary comparison tree at the hash index equal to the hash table index value created for the ACL Rule Under Consideration, illustrated is that the process proceeds to method step 608 wherein is shown that New Compare Node Pointer is set to point to the leftmost node of the binary comparison tree for the Rule Under Consideration (e.g., TCP if the Rule Under Consideration is assumed to be the rule constructed for the second rule in the example set forth in Figures 7A-7D12, below) and New Compare Node Field Type is set equal to the type of field utilized by the node pointed to by the New Compare Node Pointer (e.g., type of field is "protocol id." if the Rule Under Consideration is assumed to be the second rule in the example set forth in Figures 7A-7D12, below). Thereafter, shown in method step 610 is that Old Compare Node Pointer is set to the leftmost node (or root) of the pre-existing binary comparison tree already present at the hash index (e.g., TCP if it is assumed that the pre-existing binary comparison tree is the tree for the first rule constructed, and thereafter appended at index 0000, in the example set forth in Figures 7A-7D12, below) and that Old Compare Node Field Type is set equal to the type of field utilized by the node pointed at by the Old Compare Node Pointer (e.g., type of field

is "protocol id." if it is assumed that the Pre-Existing binary comparison tree is the tree for the first rule constructed, and thereafter appended at index 0000, in the example set forth in Figures 7A-7D12, below).

Thereafter, the process proceeds to method step 612 wherein is depicted the inquiry as to whether New Compare Node Field Type is the same as Old Compare Node Field Type (continuing with the example involving the first rule and second rule, below, both would be of type "protocol id."). If the inquiry depicted in method step 612 yields a determination that New Compare Node Field Type is the same as Old Compare Node Field Type, then the process proceeds to method step 614, wherein is shown the inquiry as to whether the value of the field utilized by the node pointed at by the New Compare Node Pointer is a subset (the term subset here means a further subdivision of an overall network concept associated with the field; for example, "source port > 50" would be considered a "subset" of "source port > 20") of the value of the field utilized by the node pointed at by the Old Compare Node Pointer. If the inquiry shown in method step 614 yields a determination that the value of the field utilized by the node pointed at by the New Compare Node Pointer is a subset of value of the field utilized by the node pointed at by the Old Compare Node Pointer, then the process proceeds to method step 616, wherein is depicted that a Next New Node Pointer is set to point to the node at the end of the match branch of the node pointed to by the New Compare Node Pointer (i.e., the next node on the match branch of the binary comparison tree for the Rule Under Consideration). Thereafter, method step 618 illustrates that New Compare Node Pointer is reset to be equal to the Next New Node Pointer.

Thereafter, the process proceeds to method step 620, wherein is depicted that Next Old Node Pointer is set to point to the node at the end of the match branch the node pointed to by the Old Compare Node Pointer (i.e., the next node on the match branch of the binary comparison tree for the Pre-Existing tree at hash table index). Thereafter, method step 622 illustrates that Old Compare Node Pointer is reset to be equal to the Next Old Node Pointer. Thereafter, the process proceeds to method step 612 and proceeds from that point.

If the inquiry shown in method step 614 yields a determination that the value of the field utilized by the node pointed at by the New Compare Node Pointer is NOT a subset of value of the field utilized by the node pointed at by the Old Compare Node Pointer, then

the process proceeds to method step 624 which shows the inquiry as to whether the value of the node on the miss branch of the node pointed to by Old Compare Node Pointer is a stop node (e.g., no further nodes extend from the node on the miss branch). If the inquiry shown in method step 624 yields a determination that the value of the field utilized by node on the miss branch of the node pointed to by Old Compare Node Pointer equates to a "stop node," method step 626 shows that the node on the miss branch of node pointed to by Old Compare Node Pointer is replaced with the node pointed at by New Compare Node Pointer (i.e., the New Compare Node Value is appended onto the Pre-Existing Binary Compare Tree). Thereafter, method step 628 depicts that a Default Deny Node Value is appended to the miss branch of the node just appended to the pre-existing hash table tree (i.e., the node pointed to by the New Compare Node pointer) as was discussed in relation to method step 626. Subsequently, the process proceeds to method step 630 and stops.

If the inquiry shown in method step 624 yields a determination that the value of the field utilized by node on the miss branch of the node pointed to by Old Compare Node Pointer DOES NOT equate to a "stop node," method step 632 shows that Next Old Node Pointer is set to point to the node at the end of the miss branch of node pointed to by the Old Compare Node Pointer (i.e., the next node on the match branch of the binary comparison tree for the Pre-Existing tree at hash table index). Thereafter, method step 634 illustrates that Old Compare Node Value is reset to be the value of the Next Old Node Pointer. Thereafter, the process proceeds to 624 and proceeds from that point.

If the inquiry depicted in method step 612 yields a determination that New Compare Node Field Type is NOT the same as Old Compare Node Field Type, then the process proceeds to method step 636, which shows that a New Stored Node Pointer is set to be equal to the New Compare Node Pointer. Thereafter, method step 638 depicts that Old Stored Node Pointer is set to be equal to Old Compare Node Pointer. Thereafter, method step 640 illustrates the inquiry as to whether the value of the node on the miss branch of the node pointed to by Old Compare Node equates to a "stop node." If the inquiry illustrated in method step 640 yields a determination that the value of the node on the miss branch of the node pointed to by Old Compare Node equates to a "stop node," the process proceeds to method step 642 which shows the addition of a tree residual, starting from the node of the binary comparison tree for the Rule Under Consideration,

pointed at by New Compare Node Pointer, at the end of the miss branch of the Old Compare Node, which is a node in the Hash Table Tree; that is, the node having a value equating to a stop node is replaced with the remainder of the binary tree for the Rule Under Consideration, where the first replacement node utilized is that node pointed at by the New Compare Node Pointer.

Thereafter, the process proceeds to method step 644 which depicts that Next Old Node Pointer is set to point to the node at the end of the match branch of Old Stored Node (we are doing this because if the field type does not match, part of the binary comparison tree created for the Rule Under Consideration must be appended to both the miss and match branch of the pre-existing hash table tree since the ACL Rule(s) encoded by the pre-existing hash table tree do not utilize the field type which is utilized by the binary tree constructed for the current Rule Under Consideration). Thereafter, method step 646 illustrates that Old Compare Node Pointer is reset to equal the Next Old Node Pointer. Thereafter, method step 648 depicts the inquiry as to whether the value of the node pointed at by Old Compare Node Pointer equates to a "stop value" (e.g., default deny, transmit packet, deny packet, etc.). In the event that the inquiry illustrated by method step 648 yields a determination that the value of the node pointed at by Old Compare Node Pointer equates to a stop value, the process proceeds to method step 649 wherein is illustrated the addition of a tree residual, starting from the node of the binary comparison tree for the Rule Under Consideration, pointed at by New Compare Node Pointer, at the end of the match branch of the Old Compare Node, which is a node in the Hash Table Tree; that is, the node having a value equating to a stop node is replaced with the remainder of the binary tree for the Rule Under Consideration, where the first replacement node utilized is that node pointed at by the New Compare Node Pointer. Thereafter, the process proceeds to method step 630 and stops. In the event that the inquiry illustrated by method step 648 yields a determination that the value of the node pointed at by Old Compare Node Pointer does NOT equate to a stop value, the process proceeds to method step 652 wherein is depicted that the New Compare Node Pointer is set to point to the node pointed at by the New Stored Node Pointer -- the reason that this pointer is not advanced at this method step is that now the process is going to append at least part of the binary Rule Under Consideration to the match branch of the pre-existing tree where the field types did not match. Thereafter, the process proceeds to method step 612 and continues from that point.

If the inquiry illustrated in method step 640 yields a determination that the node on the miss branch of the node pointed to by the Old Compare Node Pointer does NOT equate to a stop node, the process proceeds to method step 654 which shows that Next Old Node Pointer is set to point to the node at the end of the miss branch of the node pointed to by the Old Compare Node Pointer (i.e., the next node on the match branch of the binary comparison tree for the Pre-Existing tree at hash table index). Thereafter, method step 656 illustrates that the Old Compare Node Pointer is reset to equal the Next Old Node Pointer. Thereafter, the process proceeds to method step 640 and continues from that point.

With reference now to Figures 7A-7D12, shown is an example of the creation of a Hash-Table-Balancing Bit Selection Vector, and the subsequent creation of Balanced Hash Table of ABCTs such as were referred to in the flowcharts above. Figure 7A depicts an obtained hypothetical complete, ordered, ACL (e.g., the ACL referenced in Figures 1-6, above). The right-hand column of Figure 7A contains examples of the coded versions of ACL Rules which are typically utilized within an ACL. The left-hand column gives a plain English explanation of what the rules in the corresponding right hand columns mean.

Referring now to Figure 7B, depicted is an example of the creation of an exemplar bit string (e.g., such as described in relation to method step 204, above) based upon the fields utilized by the ACL rules illustrated in the right hand column of Figure 7A, and the subsequent use of the exemplar bit string to construct bit strings for each ACL Rule in the ACL illustrated in the right-hand column of Figure 7A (e.g., such as described in relation to method step 206, above). Also shown is that, for sake of illustration and ease of counting, the term "bit positions" -- as used in the examples of Figures 7B-7D12 -- will include the "periods" shown in the constructed bit strings of Figure 7B, although those skilled in art will recognize that in practice such periods are not typically counted as bit positions. To reinforce this convention, illustrated immediately below the created bit strings of Figure 7B is an illustration of the "bit position" within each constructed bit string, as that term is subsequently used in the following Figures 7C-7D12.

Referring now to Figure 7C1-7C5, illustrated is an example of the creation of a Hash-Table-Balancing Bit Selection Vector (e.g.,

such as that referenced in Figures 3A and 3B, above). The right-hand columns of Figures 7C1-7C5 depict the actual use and manipulation of quantities utilized and described in relation to Figures 3A and 3B above, while the left-hand columns corresponding with (i.e., in the same row as) the right-hand columns describe, in words, what is transpiring in the right-hand column. The example ends with specification of the Hash-Table-Balancing Bit Selection Vector.

With reference now to Figures 7D1-7D12, shown is an example of the creation of a Balanced Hash Table of ABCTs, such as was referenced in relation to Figure 1A. Figure 7D1 depicts the creation (e.g., such as that described in relation to Figure 5, above) of a Binary Comparison Rule for the first-in-sequence rule within the ACL depicted in Figure 7A. Figures 7D1-7D2 illustrate the addition of the binary comparison tree created for the first-in-sequence rule to a hash table at the index position dictated by the Hash-Table-Balancing Bit Selection Vector.

Figure 7D3 depicts the creation (e.g., such as that described in relation to Figure 5, above) of a Binary Comparison Rule for the second-in-sequence rule within the ACL depicted in Figure 7A. Figures 7D3-7D4 illustrate the addition of the binary comparison tree created for the second-in-sequence rule to a hash table at the index position dictated by the Hash-Table-Balancing Bit Selection Vector.

Figure 7D5 depicts the creation (e.g., such as that described in relation to Figure 5, above) of a Binary Comparison Rule for the third-in-sequence rule within the ACL depicted in Figure 7A. Figures 7D5-7D6 illustrate the addition of the binary comparison tree created for the third-in-sequence rule to a hash table at the index position dictated by the Hash-Table-Balancing Bit Selection Vector.

Figure 7D7 depicts the creation (e.g., such as that described in relation to Figure 5, above) of a Binary Comparison Rule for the fourth-in-sequence rule within the ACL depicted in Figure 7A. Figures 7D7-7D8 illustrate the addition of the binary comparison tree created for the fourth-in-sequence rule to a hash table at the index position dictated by the Hash-Table-Balancing Bit Selection Vector.

Figure 7D9 depicts the creation (e.g., such as that described in relation to Figure 5, above) of a Binary Comparison Rule for the fifth-in-sequence rule within the ACL depicted in Figure 7A. Figures 7D9-7D10 illustrate the addition of the binary comparison tree

created for the fifth-in-sequence rule to a hash table at the index position dictated by the Hash-Table-Balancing Bit Selection Vector.

Figure 7D11 depicts the creation (e.g., such as that described in relation to Figure 5, above) of a Binary Comparison Rule for the sixth-in-sequence rule within the ACL depicted in Figure 7A. Figures 7D11-7D12 illustrate the addition of the binary comparison tree created for the sixth-in-sequence rule to a hash table at the index position dictated by the Hash-Table-Balancing Bit Selection Vector.

The foregoing detailed description has set forth various embodiments of the present invention via the use of block diagrams, flowcharts, and examples. It will be understood as notorious by those within the art that each block diagram component, flowchart step, and operations and/or components illustrated by the use of examples can be implemented, individually and/or collectively, by a wide range of hardware, software, firmware, or any combination thereof. In one embodiment, the present invention may be implemented via Application Specific Integrated Circuits (ASICs). However, those skilled in the art will recognize that the embodiments disclosed herein, in whole or in part, can be equivalently implemented in standard Integrated Circuits, as a computer program running on a computer, as firmware, or as virtually any combination thereof and that designing the circuitry and/or writing the code for the software or firmware would be well within the skill of one of ordinary skill in the art in light of this disclosure. In addition, those skilled in the art will appreciate that the mechanisms of the present invention are capable of being distributed as a program product in a variety of forms, and that an illustrative embodiment of the present invention applies equally regardless of the particular type of signal bearing media used to actually carry out the distribution. Examples of a signal bearing media include but are not limited to the following: recordable type media such as floppy disks, hard disk drives, CD ROMs, digital tape, and transmission type media such as digital and analogue communication links using TDM or IP based communication links (e.g., packet links).

A more-preferred embodiment is set forth in the body of the present patent application. As noted above, the more-preferred embodiment may be implemented by virtually any combination of hardware, software, and/or firmware. A less-preferred embodiment is set forth in the accompanying appendix. The hardware and software aspects of this less-preferred embodiment are merely exemplary, and

those skilled in the art will recognize that the less-preferred embodiment can itself be implemented by virtually any combination of hardware, software, and/or firmware. The less-preferred embodiment is in no way intended to limit or apply to the more-preferred embodiment.

The above description is intended to be illustrative of the invention and should not be taken to be limiting. Other embodiments within the scope of the present invention are possible. Those skilled in the art will readily implement the steps necessary to provide the structures and the methods disclosed herein, and will understand that the process parameters and sequence of steps are given by way of example only and can be varied to achieve the desired structure as well as modifications that are within the scope of the invention. Variations and modifications of the embodiments disclosed herein may be made based on the description set forth herein, without departing from the spirit and scope of the invention as set forth in the following claims.

Other embodiments are within the following claims.

While particular embodiments of the present invention have been shown and described, it will be obvious to those skilled in the art that, based upon the teachings herein, changes and modifications may be made without departing from this invention and its broader aspects and, therefore, the appended claims are to encompass within their scope all such changes and modifications as are within the true spirit and scope of this invention. Furthermore, it is to be understood that the invention is solely defined by the appended claims. It will be understood by those within the art that if a specific number of an introduced claim element is intended, such an intent will be explicitly recited in the claim, and in the absence of such recitation no such limitation is present. For non-limiting example, as an aid to understanding, the following appended claims may contain usage of the introductory phrases "at least one" and "one or more" to introduce claim elements. However, the use of such phrases should not be construed to imply that the introduction of a claim element by the indefinite articles "a" or "an" limits any particular claim containing such introduced claim element to inventions containing only one such element, even when same claim includes the introductory phrases "one or more" or "at least one" and indefinite articles such as "a" or "an"; the same holds true for the use of definite articles used to introduce claim elements.

CLAIMS

1. A method comprising:
receiving at least one packet; and
disposing of the received at least one packet in response to a walk of a Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding an Access Control List.

2. The method of Claim 1, wherein said disposing of the received at least one packet in response to a walk of a Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding an Access Control List further includes:
constructing a hash table index value from one or more bit positions, within the received at least one packet, pointed at by one or more pointers of a Hash-Table-Balancing Bit Selection Vector; and
walking a binary comparison tree associated with the constructed hash table index value.

3. The method of Claim 1, further comprising:
converting the Access Control List to the Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding the Access Control List.

4. The method of Claim 3, wherein said converting the Access Control List to the Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding the Access Control List further includes:
creating a binary comparison tree for at least one Access Control List rule in the Access Control List.

5. The method of Claim 4, wherein said creating a binary comparison tree for at least one Access Control List rule further includes:
creating at least one node, having at least one miss branch and at least one match branch, for at least one packet header field utilized by the at least one Access Control List Rule in the Access Control List.

6. The method of Claim 3, wherein said converting the Access Control List to the Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding the Access Control List further includes:

inserting at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index.

7. The method of Claim 6, wherein said inserting at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index further includes:

generating a hash table index value for the at least one Access Control List rule; and

inserting the at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index which is equal to the generated hash table index value.

8. The method of Claim 7, wherein said inserting the at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index which is equal to the generated hash table index value further includes:

inserting, in its entirety, the binary comparison tree constructed for the at least one Access Control List rule into the hash table entry pointed at by the hash table index in response to a determination that no pre-existing binary comparison tree is resident within the hash table entry.

9. The method of Claim 7, wherein said inserting the at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index which is equal to the generated hash table index value further includes:

inserting at least one node of the binary comparison tree constructed for the at least one Access Control List rule into the hash table entry pointed at by the hash table index in response to a determination that a pre-existing binary comparison tree is resident within the hash table entry.

10. The method of Claim 7, wherein said generating a hash table index value for the at least one Access Control List rule further includes:

constructing the hash table index value from the contents of one or more packet headers utilized by the at least one Access Control List rule in the Access Control List.

11. The method of Claim 10, wherein said constructing the hash table index value from the contents of one or more packet headers utilized by the at least one Access Control List rule in the Access Control List further includes:

constructing the hash table index value from the contents of the one or more packet header bit positions pointed at by one or more pointers of a Hash-Table-Balancing Bit Selection Vector.

12. The method of Claim 11, wherein said constructing the hash table index value from the contents of the one or more packet header bit positions pointed at by one or more pointers of a Hash-Table-Balancing Bit Selection Vector further includes:

constructing the Hash-Table-Balancing Bit Selection Vector.

13. The method of Claim 12, wherein said constructing the Hash-Table-Balancing Bit Selection Vector further includes:

defining one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions in one or more packet header fields utilized by one or more rules of the Access Control List.

14. The method of Claim 13, wherein said defining one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions in one or more packet header fields utilized by one or more rules of the Access Control List further includes:

defining the one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions, which appear relatively most frequently, within the one or more packet header fields utilized by the one or more Rules of the Access Control List.

15. The method of Claim 13, wherein said defining one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions in one or more packet header fields utilized by one or more rules of the Access Control List further includes:

defining the one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions, whose contents have relatively equal variation between logical one and logical zero, within the one or more packet header fields utilized by the one or more Rules of the Access Control List.

16. A system comprising:
means for receiving at least one packet; and
means for disposing of the received at least one packet in response to a walk of a Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding an Access Control List.

17. The system of Claim 16, wherein said means for disposing of the received at least one packet in response to a walk of a Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding an Access Control List further includes:
means for constructing a hash table index value from one or more bit positions, within the received at least one packet, pointed at by one or more pointers of a Hash-Table-Balancing Bit Selection Vector; and
means for walking a binary comparison tree associated with the constructed hash table index value.

18. The system of Claim 16, further comprising:
means for converting the Access Control List to the Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding the Access Control List.

19. The system of Claim 18, wherein said means for converting the Access Control List to the Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding the Access Control List further includes:
means for creating a binary comparison tree for at least one Access Control List rule in the Access Control List.

20. The system of Claim 19, wherein said means for creating a binary comparison tree for at least one Access Control List rule further includes:
means for creating at least one node, having at least one miss branch and at least one match branch, for at least one packet header field utilized by the at least one Access Control List rule in the Access Control List.

21. The system of Claim 18, wherein said means for converting the Access Control List to the Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding the Access Control List further includes:

means for inserting at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index.

22. The system of Claim 21, wherein said means for inserting at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index further includes:

means for generating a hash table index value for the at least one Access Control List rule; and

means for inserting the at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index which is equal to the generated hash table index value.

23. The system of Claim 22, wherein said means for inserting the at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index which is equal to the generated hash table index value further includes:

means for inserting, in its entirety, the binary comparison tree constructed for the at least one Access Control List Rule into the hash table entry pointed at by the hash table index in response to a determination that no pre-existing binary comparison tree is resident within the hash table entry.

24. The system of Claim 22, wherein said means for inserting the at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index which is equal to the generated hash table index value further includes:

means for inserting at least one node of the binary comparison tree constructed for the at least one Access Control List rule into the hash table entry pointed at by the hash table index in response to a determination that a pre-

existing binary comparison tree is resident within the hash table entry.

25. The system of Claim 22, wherein said means for generating a hash table index value for the at least one Access Control List rule further includes:

means for constructing the hash table index value from the contents of one or more packet headers utilized by the at least one Access Control List rule in the Access Control List.

26. The system of Claim 25, wherein said means for constructing the hash table index value from the contents of one or more packet headers utilized by the at least one Access Control List rule in the Access Control List further includes:

means for constructing the hash table index value from the contents of the one or more packet header bit positions pointed at by one or more pointers of a Hash-Table-Balancing Bit Selection Vector.

27. The system of Claim 26, wherein said means for constructing the hash table index value from the contents of the one or more packet header bit positions pointed at by one or more pointers of a Hash-Table-Balancing Bit Selection Vector further includes:

means for constructing the Hash-Table-Balancing Bit Selection Vector.

28. The system of Claim 27, wherein said means for constructing the Hash-Table-Balancing Bit Selection Vector further includes:

means for defining one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions in one or more packet header fields utilized by one or more rules of the Access Control List.

29. The system of Claim 28, wherein said means for defining one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions in one or more packet header fields utilized by one or more rules of the Access Control List further includes:

means for defining the one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions, which appear relatively most frequently, within the one or more packet header fields utilized by the one or more Rules of the Access Control List.

30. The system of Claim 29, wherein said means for defining one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions in one or more packet header fields utilized by one or more rules of the Access Control List further includes:

means for defining the one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions, whose contents have relatively equal variation between logical one and logical zero, within the one or more packet header fields utilized by the one or more Rules of the Access Control List.

31. The system of Claim 16, further comprising:

signal bearing media bearing

said means for receiving at least one packet, and

said means for disposing of the received at least one packet in response to a walk of a Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding an Access Control List.

32. The system of Claim 31, wherein said signal bearing media further includes:

recordable media.

33. The system of Claim 31, wherein said signal bearing media further includes:

transmission media.

34. The system of Claim 16, wherein the system further includes:

a network station.

35. A program product comprising:
signal bearing media bearing
means for receiving at least one packet, and
means for disposing of the received at least one packet in response to a walk of a Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding an Access Control List.

36. The program product of Claim 35, wherein said signal bearing media further includes:
recordable media.

37. The program product of Claim 35, wherein said signal bearing media further includes:
transmission media.

38. The program product of Claim 35, wherein said means for disposing of the received at least one packet in response to a walk of a Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding an Access Control List further includes:
means for constructing a hash table index value from one or more bit positions, within the received at least one packet, pointed at by one or more pointers of a Hash-Table-Balancing Bit Selection Vector; and
means for walking a binary comparison tree associated with the constructed hash table index value.

39. The program product of Claim 35, further comprising:
means for converting the Access Control List to the Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding the Access Control List.

40. The program product of Claim 39, wherein said means for converting the Access Control List to the Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding the Access Control List further includes:
means for creating a binary comparison tree for at least one Access Control List rule in the Access Control List.

41. The program product of Claim 40, wherein said means for creating a binary comparison tree for at least one Access Control List rule further includes:

means for creating at least one node, having at least one miss branch and at least one match branch, for at least one packet header field utilized by the at least one Access Control List rule in the Access Control List.

42. The program product of Claim 39, wherein said means for converting the Access Control List to the Balanced Hash Table of Access Control List Binary Comparison Trees, the Table encoding the Access Control List further includes:

means for inserting at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index.

43. The program product of Claim 42, wherein said means for inserting at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index further includes:

means for generating a hash table index value for the at least one Access Control List rule; and

means for inserting the at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index which is equal to the generated hash table index value.

44. The program product of Claim 43, wherein said means for inserting the at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index which is equal to the generated hash table index value further includes:

means for inserting, in its entirety, the binary comparison tree constructed for the at least one Access Control List Rule into the hash table entry pointed at by the hash table index in response to a determination that no pre-existing binary comparison tree is resident within the hash table entry.

45. The program product of Claim 43, wherein said means for inserting the at least a part of a binary comparison tree constructed for at least one Access Control List rule into a hash table entry pointed at by a hash table index which is equal to the generated hash table index value further includes:

means for inserting at least one node of the binary comparison tree constructed for the at least one Access Control List rule into the hash table entry pointed at by the hash table index in response to a determination that a pre-existing binary comparison tree is resident within the hash table entry.

46. The program product of Claim 43, wherein said means for generating a hash table index value for the at least one Access Control List rule further includes:

means for constructing the hash table index value from the contents of one or more packet headers utilized by the at least one Access Control List rule in the Access Control List.

47. The program product of Claim 46, wherein said means for constructing the hash table index value from the contents of one or more packet headers utilized by the at least one Access Control List rule in the Access Control List further includes:

means for constructing the hash table index value from the contents of the one or more packet header bit positions pointed at by one or more pointers of a Hash-Table-Balancing Bit Selection Vector.

48. The program product of Claim 47, wherein said means for constructing the hash table index value from the contents of the one or more packet header bit positions pointed at by one or more pointers of a Hash-Table-Balancing Bit Selection Vector further includes:

means for constructing the Hash-Table-Balancing Bit Selection Vector.

49. The program product of Claim 48, wherein said means for constructing the Hash-Table-Balancing Bit Selection Vector further includes:

means for defining one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions in one or more packet header fields utilized by one or more rules of the Access Control List.

50. The program product of Claim 49, wherein said means for defining one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions in one or more packet header fields utilized by one or more rules of the Access Control List further includes:

means for defining the one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions, which appear relatively most frequently, within the one or more packet header fields utilized by the one or more Rules of the Access Control List.

51. The program product of Claim 50, wherein said means for defining one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions in one or more packet header fields utilized by one or more rules of the Access Control List further includes:

means for defining the one or more pointers of the Hash-Table-Balancing Bit Selection Vector to point to one or more bit positions, whose contents have relatively equal variation between logical one and logical zero, within the one or more packet header fields utilized by the one or more Rules of the Access Control List.

IMPLEMENTING ACCESS CONTROL LISTS USING A BALANCED HASH TABLE OF ACCESS CONTROL LIST BINARY COMPARISON TREES

Faisal Haq
Hari K. Lalgudi

5

ABSTRACT OF THE DISCLOSURE

A method and system for implementing Access Control Lists (ACLs) using a Balanced Hash Table of ACL Binary Comparison Trees (ABCTs), where the Balanced Hash Table of ABCTs encodes the replaced ACL. In one embodiment, the method includes but is not limited to receiving at least one packet, and disposing of the received at least one packet in response to a walk of a Balanced Hash Table of ABCTs, where the Balanced Hash Table of ABCTs encodes an Access Control List. In another embodiment, the method further includes converting the Access Control List to the Balanced Hash Table of ABCTs, the Balanced Hash Table of ABCTs encoding the Access Control List. In one embodiment, the system includes means for receiving at least one packet, and means for disposing of the received at least one packet in response to a walk of a Balanced Hash Table of ABCTs, where the Balanced Hash Table of ABCTs encodes an Access Control List. In another embodiment, the system further includes means for converting the Access Control List to the Balanced Hash Table of ABCTs, where the Balanced Hash Table of ABCTs encodes the Access Control List.

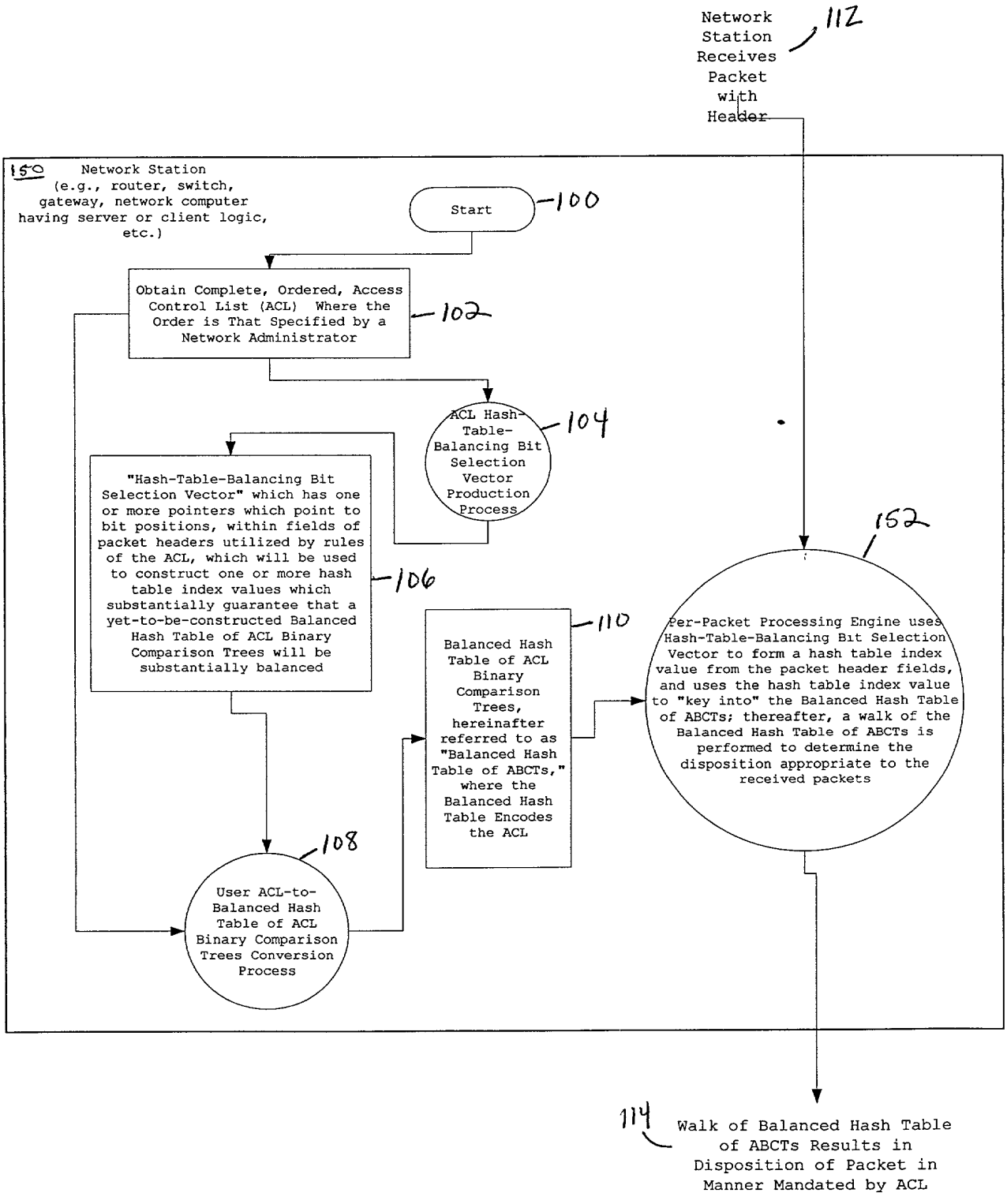

Network Station Receives Packet with Header
112
150
Network Station (e.g., router, switch, gateway, network computer having server or client logic, etc.)
Start
100
Obtain Complete, Ordered, Access Control List (ACL) Where the Order is That Specified by a Network Administrator
102
ACL Hash-Table-Balancing Bit Selection Vector Production Process
104
"Hash-Table-Balancing Bit Selection Vector" which has one or more pointers which point to bit positions, within fields of packet headers utilized by rules of the ACL, which will be used to construct one or more hash table index values which substantially guarantee that a yet-to-be-constructed Balanced Hash Table of ACL Binary Comparison Trees will be substantially balanced
106
Balanced Hash Table of ACL Binary Comparison Trees, hereinafter referred to as "Balanced Hash Table of ABCTs," where the Balanced Hash Table Encodes the ACL
110
User ACL-to-Balanced Hash Table of ACL Binary Comparison Trees Conversion Process
108
Per-Packet Processing Engine uses Hash-Table-Balancing Bit Selection Vector to form a hash table index value from the packet header fields, and uses the hash table index value to "key into" the Balanced Hash Table of ABCTs; thereafter, a walk of the Balanced Hash Table of ABCTs is performed to determine the disposition appropriate to the received packets
152
114
Walk of Balanced Hash Table of ABCTs Results in Disposition of Packet in Manner Mandated by ACL

FIG. 1A

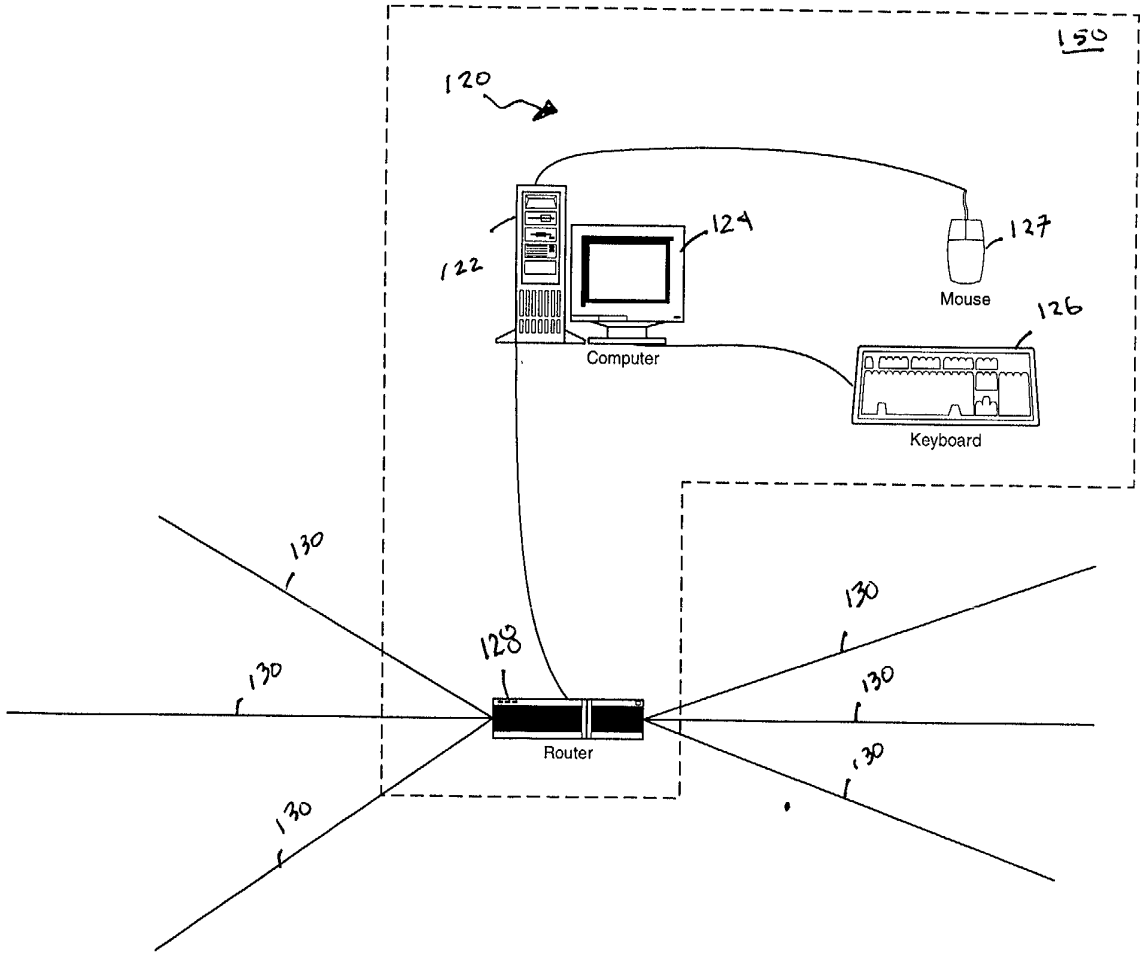


FIG. 1B

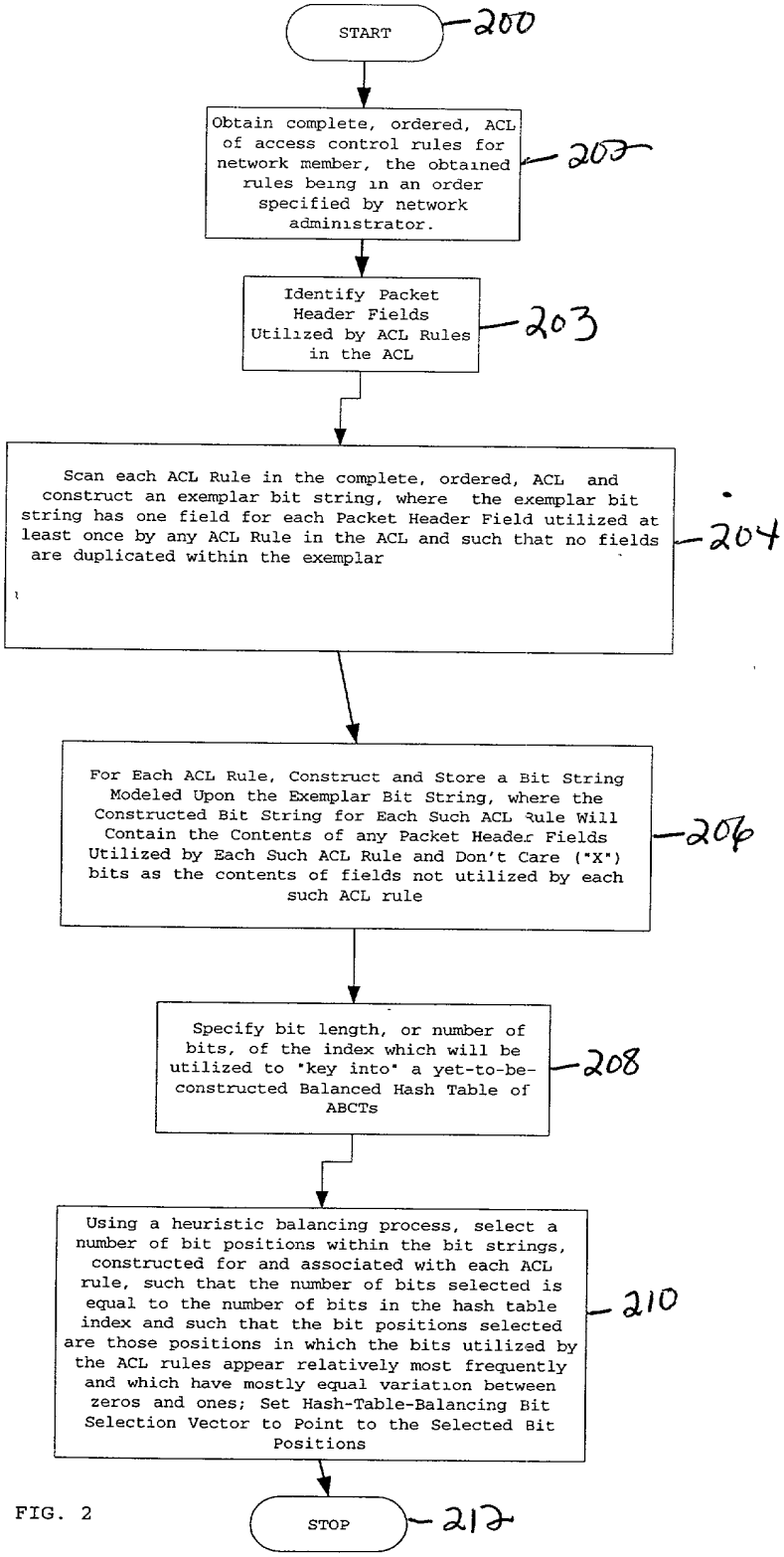

START
200
Obtain complete, ordered, ACL of access control rules for network member, the obtained rules being in an order specified by network administrator.
202
Identify Packet Header Fields Utilized by ACL Rules in the ACL
203
Scan each ACL Rule in the complete, ordered, ACL and construct an exemplar bit string, where the exemplar bit string has one field for each Packet Header Field utilized at least once by any ACL Rule in the ACL and such that no fields are duplicated within the exemplar
204
For Each ACL Rule, Construct and Store a Bit String Modeled Upon the Exemplar Bit String, where the Constructed Bit String for Each Such ACL Rule Will Contain the Contents of any Packet Header Fields Utilized by Each Such ACL Rule and Don't Care ("X") bits as the contents of fields not utilized by each such ACL rule
206
Specify bit length, or number of bits, of the index which will be utilized to "key into" a yet-to-be-constructed Balanced Hash Table of ABCTs
208
Using a heuristic balancing process, select a number of bit positions within the bit strings, constructed for and associated with each ACL rule, such that the number of bits selected is equal to the number of bits in the hash table index and such that the bit positions selected are those positions in which the bits utilized by the ACL rules appear relatively most frequently and which have mostly equal variation between zeros and ones; Set Hash-Table-Balancing Bit Selection Vector to Point to the Selected Bit Positions
210
STOP
212

FIG. 2

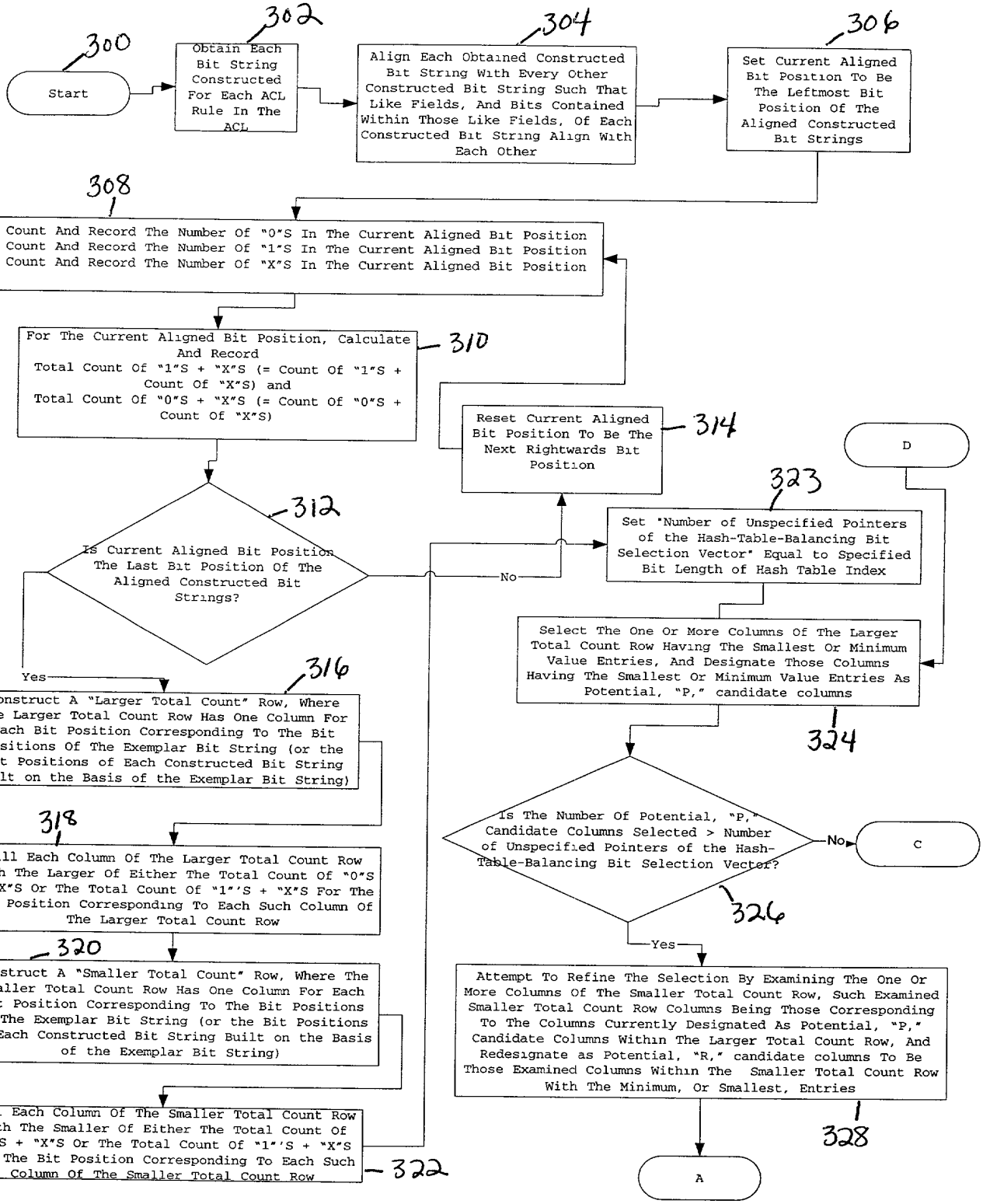

300
Start
302
Obtain Each Bit String Constructed For Each ACL Rule In The ACL
304
Align Each Obtained Constructed Bit String With Every Other Constructed Bit String Such That Like Fields, And Bits Contained Within Those Like Fields, Of Each Constructed Bit String Align With Each Other
306
Set Current Aligned Bit Position To Be The Leftmost Bit Position Of The Aligned Constructed Bit Strings
308
Count And Record The Number Of "0"S In The Current Aligned Bit Position
Count And Record The Number Of "1"S In The Current Aligned Bit Position
Count And Record The Number Of "X"S In The Current Aligned Bit Position
For The Current Aligned Bit Position, Calculate And Record
Total Count Of "1"S + "X"S (= Count Of "1"S + Count Of "X"S) and
Total Count Of "0"S + "X"S (= Count Of "0"S + Count Of "X"S)
310
Reset Current Aligned Bit Position To Be The Next Rightwards Bit Position
314
312
Is Current Aligned Bit Position The Last Bit Position Of The Aligned Constructed Bit Strings?
No
Yes
D
323
Set "Number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector" Equal to Specified Bit Length of Hash Table Index
Select The One Or More Columns Of The Larger Total Count Row Having The Smallest Or Minimum Value Entries, And Designate Those Columns Having The Smallest Or Minimum Value Entries As Potential, "P," candidate columns
324
316
Construct A "Larger Total Count" Row, Where The Larger Total Count Row Has One Column For Each Bit Position Corresponding To The Bit Positions Of The Exemplar Bit String (or the Bit Positions of Each Constructed Bit String Built on the Basis of the Exemplar Bit String)
318
Fill Each Column Of The Larger Total Count Row With The Larger Of Either The Total Count Of "0"S + "X"S Or The Total Count Of "1"'S + "X"S For The Bit Position Corresponding To Each Such Column Of The Larger Total Count Row
320
Construct A "Smaller Total Count" Row, Where The Smaller Total Count Row Has One Column For Each Bit Position Corresponding To The Bit Positions Of The Exemplar Bit String (or the Bit Positions of Each Constructed Bit String Built on the Basis of the Exemplar Bit String)
Fill Each Column Of The Smaller Total Count Row With The Smaller Of Either The Total Count Of "0"S + "X"S Or The Total Count Of "1"'S + "X"S For The Bit Position Corresponding To Each Such Column Of The Smaller Total Count Row
322
Is The Number Of Potential, "P," Candidate Columns Selected > Number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector?
326
No
C
Yes
Attempt To Refine The Selection By Examining The One Or More Columns Of The Smaller Total Count Row, Such Examined Smaller Total Count Row Columns Being Those Corresponding To The Columns Currently Designated As Potential, "P," Candidate Columns Within The Larger Total Count Row, And Redesignate as Potential, "R," candidate columns To Be Those Examined Columns Within The Smaller Total Count Row With The Minimum, Or Smallest, Entries
328
A

FIG. 3A

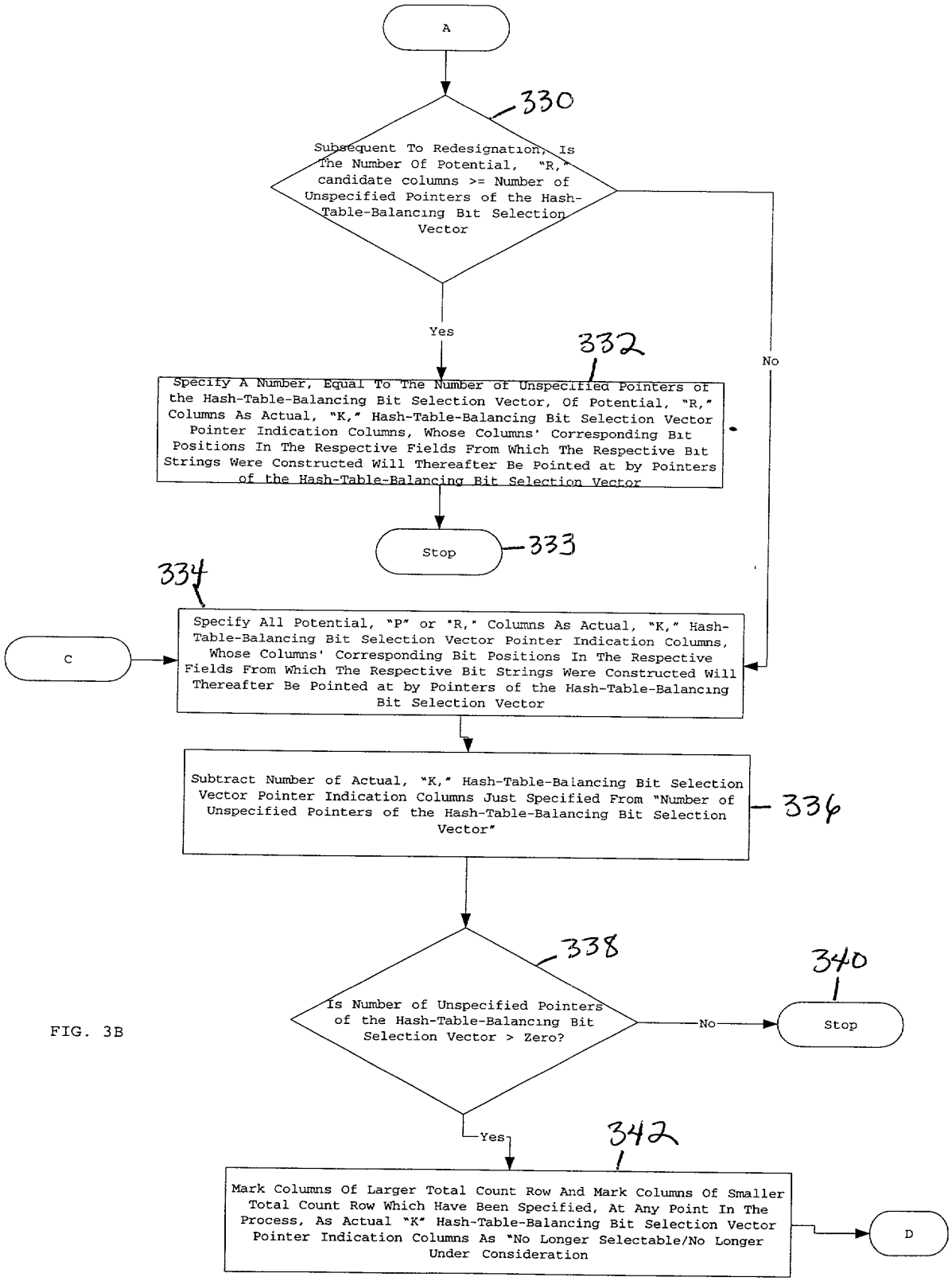

A
330
Subsequent To Redesignation, Is The Number Of Potential, "R," candidate columns >= Number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector
Yes
No
332
Specify A Number, Equal To The Number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector, Of Potential, "R," Columns As Actual, "K," Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns, Whose Columns' Corresponding Bit Positions In The Respective Fields From Which The Respective Bit Strings Were Constructed Will Thereafter Be Pointed at by Pointers of the Hash-Table-Balancing Bit Selection Vector
Stop
333
334
C
Specify All Potential, "P" or "R," Columns As Actual, "K," Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns, Whose Columns' Corresponding Bit Positions In The Respective Fields From Which The Respective Bit Strings Were Constructed Will Thereafter Be Pointed at by Pointers of the Hash-Table-Balancing Bit Selection Vector
Subtract Number of Actual, "K," Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns Just Specified From "Number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector"
336
338
340
Is Number of Unspecified Pointers of the Hash-Table-Balancing Bit Selection Vector > Zero?
No
Stop
Yes
342
Mark Columns Of Larger Total Count Row And Mark Columns Of Smaller Total Count Row Which Have Been Specified, At Any Point In The Process, As Actual "K" Hash-Table-Balancing Bit Selection Vector Pointer Indication Columns As "No Longer Selectable/No Longer Under Consideration
D

FIG. 3B

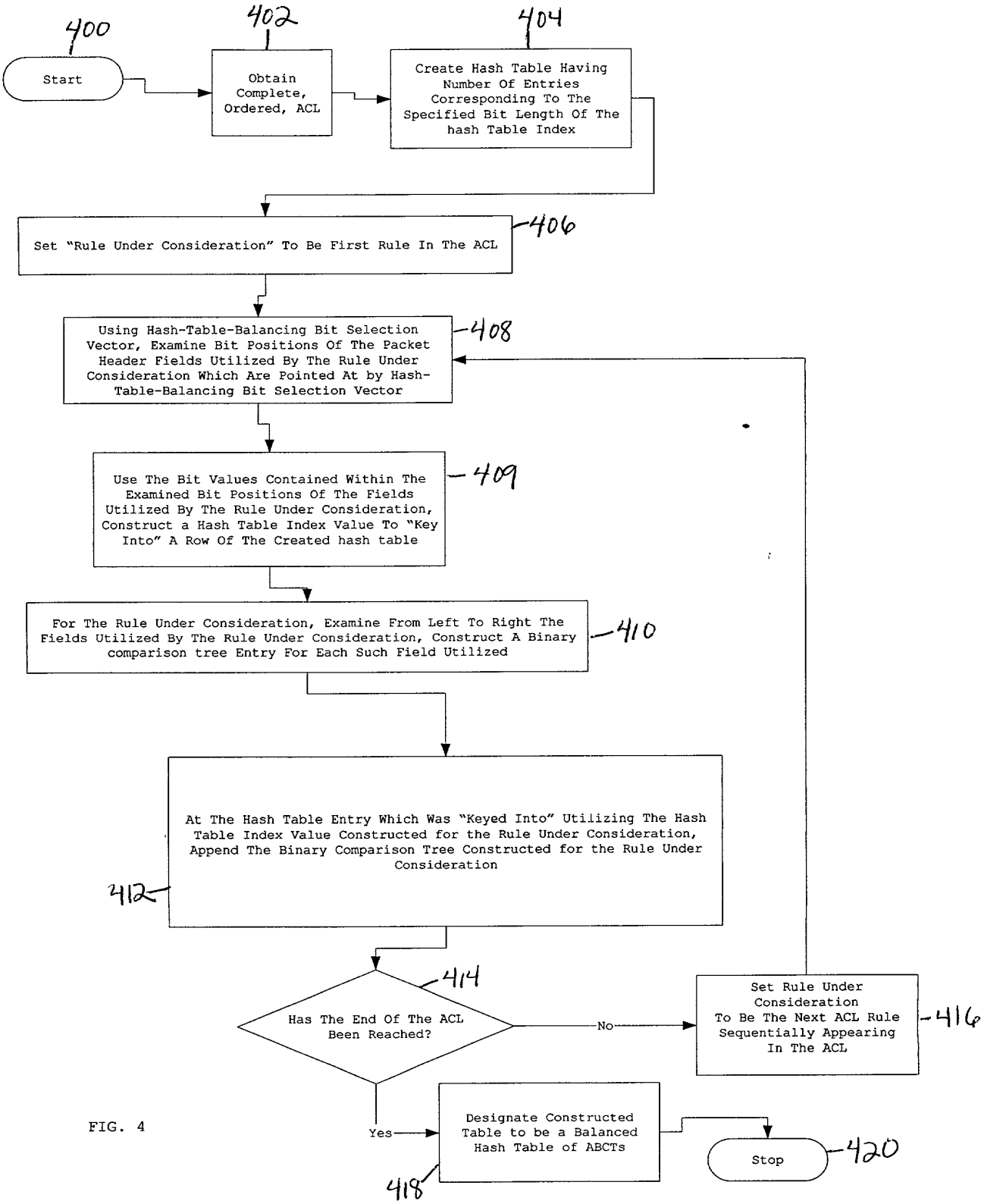

400
Start
402
Obtain Complete, Ordered, ACL
404
Create Hash Table Having Number Of Entries Corresponding To The Specified Bit Length Of The hash Table Index
Set "Rule Under Consideration" To Be First Rule In The ACL
406
Using Hash-Table-Balancing Bit Selection Vector, Examine Bit Positions Of The Packet Header Fields Utilized By The Rule Under Consideration Which Are Pointed At by Hash-Table-Balancing Bit Selection Vector
408
Use The Bit Values Contained Within The Examined Bit Positions Of The Fields Utilized By The Rule Under Consideration, Construct a Hash Table Index Value To "Key Into" A Row Of The Created hash table
409
For The Rule Under Consideration, Examine From Left To Right The Fields Utilized By The Rule Under Consideration, Construct A Binary comparison tree Entry For Each Such Field Utilized
410
At The Hash Table Entry Which Was "Keyed Into" Utilizing The Hash Table Index Value Constructed for the Rule Under Consideration, Append The Binary Comparison Tree Constructed for the Rule Under Consideration
412
Has The End Of The ACL Been Reached?
414
No
Set Rule Under Consideration To Be The Next ACL Rule Sequentially Appearing In The ACL
416
Yes
Designate Constructed Table to be a Balanced Hash Table of ABCTs
418
Stop
420

FIG. 4

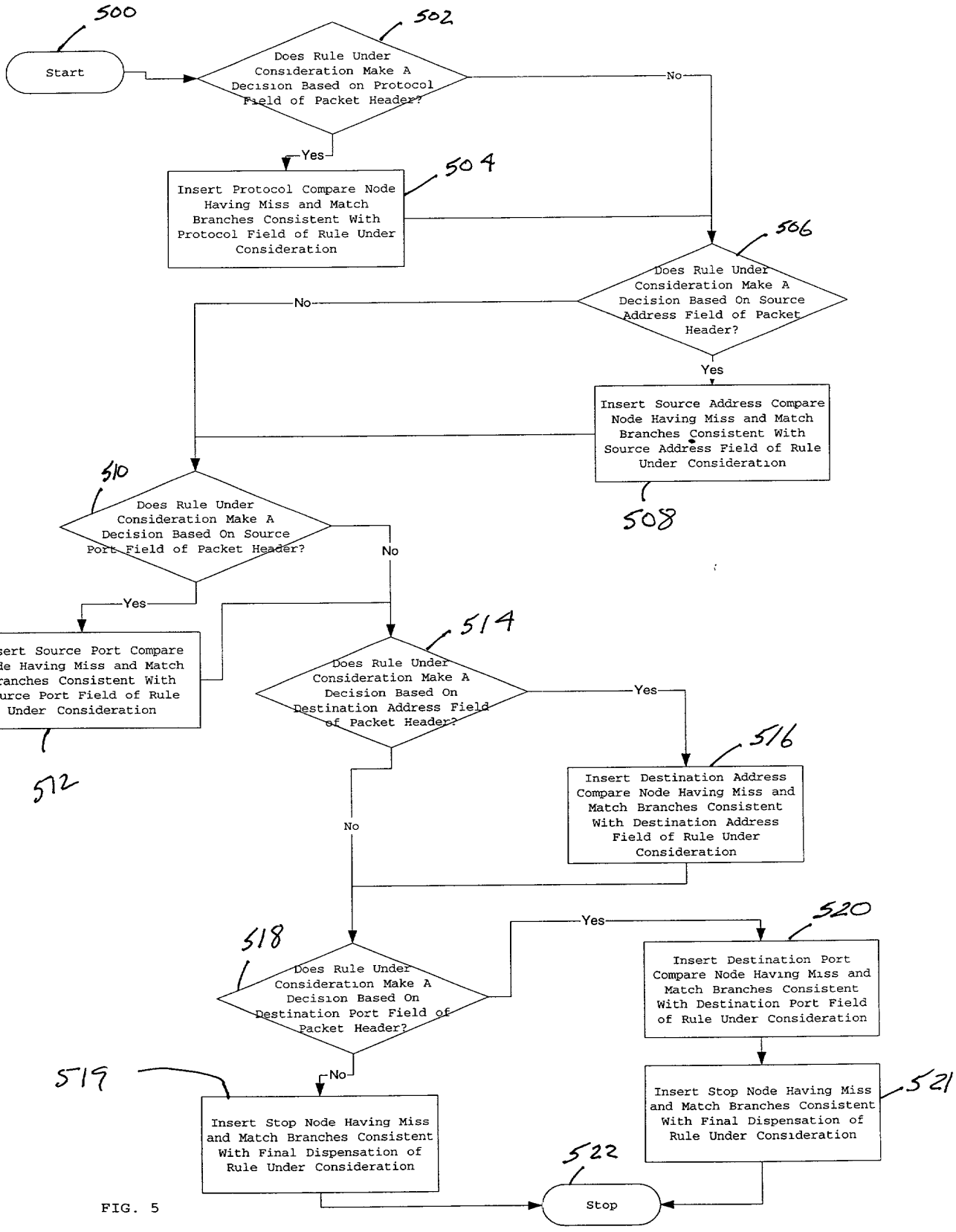

500
Start
502
Does Rule Under Consideration Make A Decision Based on Protocol Field of Packet Header?
No
Yes
504
Insert Protocol Compare Node Having Miss and Match Branches Consistent With Protocol Field of Rule Under Consideration
506
Does Rule Under Consideration Make A Decision Based On Source Address Field of Packet Header?
No
Yes
Insert Source Address Compare Node Having Miss and Match Branches Consistent With Source Address Field of Rule Under Consideration
508
510
Does Rule Under Consideration Make A Decision Based On Source Port Field of Packet Header?
No
Yes
Insert Source Port Compare Node Having Miss and Match Branches Consistent With Source Port Field of Rule Under Consideration
512
514
Does Rule Under Consideration Make A Decision Based On Destination Address Field of Packet Header?
Yes
No
516
Insert Destination Address Compare Node Having Miss and Match Branches Consistent With Destination Address Field of Rule Under Consideration
518
Does Rule Under Consideration Make A Decision Based On Destination Port Field of Packet Header?
Yes
No
520
Insert Destination Port Compare Node Having Miss and Match Branches Consistent With Destination Port Field of Rule Under Consideration
519
Insert Stop Node Having Miss and Match Branches Consistent With Final Dispensation of Rule Under Consideration
521
Insert Stop Node Having Miss and Match Branches Consistent With Final Dispensation of Rule Under Consideration
522
Stop

FIG. 5

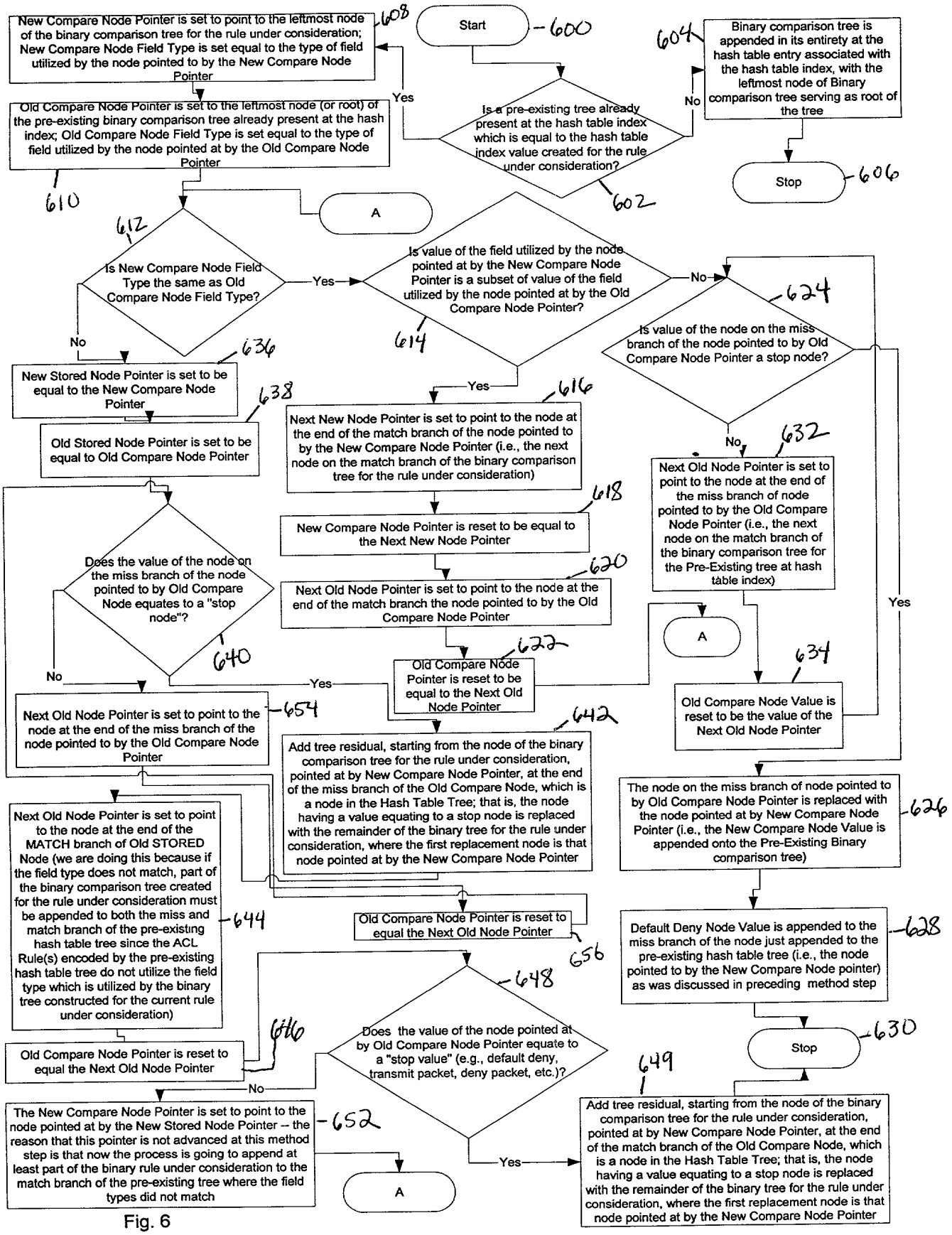

Start
600
Is a pre-existing tree already present at the hash table index which is equal to the hash table index value created for the rule under consideration?
602
Yes
No
New Compare Node Pointer is set to point to the leftmost node of the binary comparison tree for the rule under consideration; New Compare Node Field Type is set equal to the type of field utilized by the node pointed to by the New Compare Node Pointer
608
Old Compare Node Pointer is set to the leftmost node (or root) of the pre-existing binary comparison tree already present at the hash index; Old Compare Node Field Type is set equal to the type of field utilized by the node pointed at by the Old Compare Node Pointer
610
Binary comparison tree is appended in its entirety at the hash table entry associated with the hash table index, with the leftmost node of Binary comparison tree serving as root of the tree
604
Stop
606
A
Is New Compare Node Field Type the same as Old Compare Node Field Type?
612
Yes
No
Is value of the field utilized by the node pointed at by the New Compare Node Pointer is a subset of value of the field utilized by the node pointed at by the Old Compare Node Pointer?
614
Yes
No
Is value of the node on the miss branch of the node pointed to by Old Compare Node Pointer a stop node?
624
Yes
No
New Stored Node Pointer is set to be equal to the New Compare Node Pointer
636
Old Stored Node Pointer is set to be equal to Old Compare Node Pointer
638
Next New Node Pointer is set to point to the node at the end of the match branch of the node pointed to by the New Compare Node Pointer (i.e., the next node on the match branch of the binary comparison tree for the rule under consideration)
616
New Compare Node Pointer is reset to be equal to the Next New Node Pointer
618
Next Old Node Pointer is set to point to the node at the end of the match branch the node pointed to by the Old Compare Node Pointer
620
Old Compare Node Pointer is reset to be equal to the Next Old Node Pointer
622
Next Old Node Pointer is set to point to the node at the end of the miss branch of node pointed to by the Old Compare Node Pointer (i.e., the next node on the match branch of the binary comparison tree for the Pre-Existing tree at hash table index)
632
A
Old Compare Node Value is reset to be the value of the Next Old Node Pointer
634
Does the value of the node on the miss branch of the node pointed to by Old Compare Node equates to a "stop node"?
640
No
Yes
Next Old Node Pointer is set to point to the node at the end of the miss branch of the node pointed to by the Old Compare Node Pointer
654
Add tree residual, starting from the node of the binary comparison tree for the rule under consideration, pointed at by New Compare Node Pointer, at the end of the miss branch of the Old Compare Node, which is a node in the Hash Table Tree; that is, the node having a value equating to a stop node is replaced with the remainder of the binary tree for the rule under consideration, where the first replacement node is that node pointed at by the New Compare Node Pointer
642
The node on the miss branch of node pointed to by Old Compare Node Pointer is replaced with the node pointed at by New Compare Node Pointer (i.e., the New Compare Node Value is appended onto the Pre-Existing Binary comparison tree)
626
Next Old Node Pointer is set to point to the node at the end of the MATCH branch of Old STORED Node (we are doing this because if the field type does not match, part of the binary comparison tree created for the rule under consideration must be appended to both the miss and match branch of the pre-existing hash table tree since the ACL Rule(s) encoded by the pre-existing hash table tree do not utilize the field type which is utilized by the binary tree constructed for the current rule under consideration)
644
Old Compare Node Pointer is reset to equal the Next Old Node Pointer
656
Default Deny Node Value is appended to the miss branch of the node just appended to the pre-existing hash table tree (i.e., the node pointed to by the New Compare Node pointer) as was discussed in preceding method step
628
Does the value of the node pointed at by Old Compare Node Pointer equate to a "stop value" (e.g., default deny, transmit packet, deny packet, etc.)?
648
Old Compare Node Pointer is reset to equal the Next Old Node Pointer
646
Stop
630
No
Yes
The New Compare Node Pointer is set to point to the node pointed at by the New Stored Node Pointer -- the reason that this pointer is not advanced at this method step is that now the process is going to append at least part of the binary rule under consideration to the match branch of the pre-existing tree where the field types did not match
652
A
Add tree residual, starting from the node of the binary comparison tree for the rule under consideration, pointed at by New Compare Node Pointer, at the end of the match branch of the Old Compare Node, which is a node in the Hash Table Tree; that is, the node having a value equating to a stop node is replaced with the remainder of the binary tree for the rule under consideration, where the first replacement node is that node pointed at by the New Compare Node Pointer
649

Fig. 6

Simplified Example of Ordered ACL Rule Set Typically Entered by a Network Administrator

ACL Rules in an Ordered ACL Rule Set Expressed as Plain English Statements	Examples of Coded Versions of ACL Rules Which Are Typically Utilized Within an ACL Rule Set
Permit TCP protocol packets from any source IP address going to host having an IP address of 28.16.31.10 and a port identifier equal to 28.	PERMIT TCP ANY HOST 28.16.31.10 EQ 28
Deny TCP protocol packets from any source IP address going to host having an IP address of 28.16.31.10 and a port identifier greater than 23.	DENY TCP ANY HOST 28.16.31.10 GT 23
Deny UDP protocol packets from any source IP address going to host having an IP address of 30.22.12.5 and a port identifier equal to 11.	DENY UDP ANY HOST 30.22.21.5 EQ 11
Permit UDP protocol packets from any source IP address going to host having an IP address of 30.22.12.X, where X indicates any number, or "don't care".	PERMIT UDP ANY HOST 30.22.21.X
Deny all packets having source IP address of 23.20.7.0 and any destination address (indicated by address X.X.X.X, where X indicates any number, or "don't care").	DENY TCP 23.20.7.0 X.X.X.X.
Permit TCP protocol packets from any source IP address going to host having an IP address of 28.16.31.10.	PERMIT TCP ANY HOST 28.16.31.10

FIG. 7A

Example of the Creation of an Exemplar Bit String Having One Field for Each Packet Header Field Utilized By at Least One ACL Rule in the ACL Rule Set, and the Subsequent Creation of Bit Strings for Each ACL Rule in the ACL Rule Set Based on the Created Exemplar

Construct Exemplar Bit String Based On Packet Header Fields Utilized by ACL Rule Set Rules	Protocol Id .	Source Address .	Destination Address .	Destination Port	
Bit String, based on exemplar, for **ACL Rule 1** with string "01001" associated with TCP protocol for sake of example.	01001.	XXXXX.XXXXX.XXXXX.XXXXX.	11100.10000.	11111.01010.	10111
Bit String, based on exemplar, for **ACL Rule 2** with string "01001" associated with TCP protocol for sake of example.	01001.	XXXXX.XXXXX.XXXXX.XXXXX.	11100.10000.	11111.01010.	11100
Bit String, based on exemplar, for **ACL Rule 3** with string "11111" associated with UDP protocol for sake of example.	11111.	XXXXX.XXXXX.XXXXX.XXXXX.	11110.10110.	10101.00101.	01011
Bit string, based on exemplar, for **ACL Rule 4** with string "11111" associated with UDP protocol for sake of example.	11111.	XXXXX.XXXXX.XXXXX.XXXXX.	11110.10110.	10101.XXXXX.	XXXXX
Bit string, based on exemplar, for **ACL Rule 5** with string "01001" associated with TCP protocol for sake of example.	01001.	10111.10101.00111.00000.	XXXXX.XXXXX.	XXXXX.XXXXX.	XXXXX
Bit String, based on exemplar, for **ACL Rule 6** with string "01001" associated with TCP protocol for sake of example.	01001.	XXXXX.XXXXX.XXXXX.XXXXX.	11100.10000.	11111.01010.	XXXXX
Note: "Bit Position" is illustrated for sake of clarity and ease of counting herein as taking account of the periods between 5 bit fields; however, those skilled in the art will recognize that ordinarily such periods are not counted as bit positions.	00000000011111111112222222222333333333344444444445555555555 12345678901234567890123456789012345678901234567890123456789				<- Read Bit Position <- Slot Numbers Vertically. For example, the first bit position is denoted 0 0 1, the second, 2., the 0 0 third, 3, the fourth, 4, 1 the eleventh, 1, and the 5 fifty-ninth, 9.

FIG. 7B

Example of the Creation of a Bit Selection Vector

"0"Count in Each Bit Position: "X"Count in Each Bit Position:	40440.01000.01010.11000.11111.00035.05335.02020.41313.11111 00000.55555.55555.55555.55555.11111.11111.11111.22222.33333
Total of "O" + "X" Counts:	**40440.56555.56565.66555.66666.11146.16446.13131.63535.44444**
"1"Count in Each Bit Position: "X"Count in Each Bit Position:	26226.10111.10101.00111.00000.55520.50220.53535.03131.22222 00000.55555.55555.55555.55555.11111.11111.11111.22222.33333
Total of "1" + "X" Counts:	**26226.65666.65656.55666.55555.66631.61331.64645.25353.55555**
Construct a "Larger Total Count" row having one row entry corresponding to each bit position in the strings which were constructed from the ACL rules; fill each row entry with the larger of either the "Total of '0' + 'X' Counts" or "Total of '1' + 'X' Counts" for the bit position corresponding to that row entry.	46446.66666.66666.66666.66666.66646.66446.64645.65555.55555
Construct a "Smaller Total Count" row having one row entry corresponding to each bit position in the strings which were constructed from the ACL rules; fill each row entry with the smaller of either the "Total of '0' + 'X' Counts" or "Total of '1' + 'X' Counts" for the bit position corresponding to that row entry.	20226.55555.55555.55555.55555.11131.11331.13131.23333.44444
Set Number of Unspecified Pointers of Bit Selection Vector = Specified Bit Length of Hash table index	Number of Unspecified Pointers of Bit Selection Vector = 4 For sake of example, assume hash table index having a bit length of 4 is specified.
Select the row entries in the "Larger Total Count" row columns having the smallest number entries; designate the bit positions corresponding to the selected row columns as potential, "P," candidate columns which might be utilized as the pointers of the Bit Selection Vector	P PP P PP P P Note: The row columns 1, 3, 34, 39,41, and 46 of the "Larger Total Count" row had the smallest entries (i.e., the base 10 number "4"), and thus the bit positions associated with row columns 1, 3, 34, 39, 41, and 46 of the "Larger Total Count" row are designated as potential candidate bits "P."
Since there are more Potential, "P," candidate columns than number of Unspecified Pointers of Bit Selection Vector, refine the selection by Examining the columns of the Smaller Total	R RR Note: The row columns 1, 3, and 4 of the "Smaller Total Count" row, corresponding with the selected row columns of the "Larger Total Count" row, had the smallest entries (i.e., the base 10 number "2"), and thus the bit positions associated with row columns 1, 3, and 4 of the "Smaller Total Count" row are redesignated as

FIG. 7C1

Count Row, with such examined Smaller Total Count row columns being those corresponding to the Larger Total Count Row columns designated as potential, "P," candidate columns; redesignate as potential, "R," candidate columns which might be utilized as the pointers of the Bit Selection Vector, those examined Smaller Total Count row columns with the smallest number entries	candidate bits "R."
Since the number or redesignated potential candidates, "R," is less than the Number of Unspecified Pointers of Bit Selection Vector, Designate all redesignated, "R, candidates as Actual, "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by pointers of the Bit Selection Vector	K KK Note: The Number of Unspecified Pointers of Bit Selection Vector is currently equal to 4, and the number of redesignated potential candidates, "R," is 3, which is less than the Number of Unspecified Pointers of Bit Selection Vector; thus, all "R" potential candidates are specified Actual, "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by pointers of the Bit Selection Vector.
Subtract the number of Specified Actual, "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by pointers of the Bit Selection Vector, from Number of Unspecified Pointers of Bit Selection Vector	Number of Unspecified Pointers of Bit Selection Vector = Number of Unspecified Pointers of Bit Selection Vector (i.e., 4) - Number of Specified Actual, "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by pointers of the Bit Selection Vector Specified in Preceding Step(i.e., 3) = 1 pointer left unspecified
Since the number of Unspecified Pointers of Bit Selection Vector is still non-zero, Mark Specified "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by	* ** Note: Row columns 1, 3, and 4 are marked with asterisks to indicate that since the these row columns have already been designated as candidates "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by pointers of the Bit Selection Vector.

FIG. 7C2

pointers of the Bit Selection Vector with asterisks indicating that such columns are no longer selectable or under consideration, since the bit positions associated with the "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by pointers of the Bit Selection Vector have already been specified.	
Thereafter, repeat the "select the row entries in the "Larger Total Count" row having smallest number entries . . ." operation above upon the row columns which have not yet been designated as candidate "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by pointers of the Bit Selection Vector	* ** P PP P P Note: Row columns 1, 3, and 4 are marked with asterisks to indicate that since the bit positions associated with these row columns have already been designated as candidates.

FIG. 7C3

Since there are more candidates, "P," than Number of Unspecified Pointers of Bit Selection Vector(at this point, 3 pointers have been specified as "K," meaning that one additional pointer is necessary to have the pointers required to completely point out the 4 bit hash table index), repeat the refine the selection . . ." operation above	R RR R R Note: Since all entries in the "Smaller Total Count" row columns, corresponding with the selected row columns of the "Larger Total Count" row, were the same number (i.e., the base ten number "3"), all P row columns are redesignated as candidates "R."
Since after redesignation there are still more candidates "R" than the Number of Unspecified Pointers of Bit Selection Vector, all "R" candidates are deemed equally good choices; consequently, the number of Actual, "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by pointers of the Bit Selection Vector necessary to completely point out the hash table index value (i.e., in the present example, one more pointer is needed) may be selected at random from the designated "R" row columns.	K Note: Select row column 34 at random.
There are now specified Actual, "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by pointers of the Bit Selection Vector equal in number to the bit length of the hash table index; consequently, all pointers of the Bit Selection Vector, which will be utilized to point to bit positions used to form a hash table index value which will be used to "key into"	K KK K Note: These Actual, "K," Bit Selection Vector Pointer Indication Columns, whose corresponding bit positions in the respective fields from which the respective bit strings were constructed will thereafter be pointed at by pointers of the Bit Selection Vector indicate that the first, third, and fourth leftmost bit positions within the "protocol ID" field, and the fourth leftmost bit positions within the "destination address" field will be utilized as the hash table index bits.

FIG. 7C4

the hash table, have been fully specified.	
Definition of the Bit Selection Vector	Bit Selection Vector = [pointer to first leftmost bit position within the "protocol ID" field; pointer to third leftmost bit position within the "protocol ID" field; pointer to fourth leftmost bit position within the "protocol ID" field; pointer to fourth leftmost bit position within the "destination address" field]

FIG. 7C5

Example Showing the Construction of Balanced Hash Table Of ACL Binary Comparison Trees

Example Showing the Creation of a Binary Comparison Tree for First In Sequence ACL Rule in Rule Set

PERMIT TCP ANY HOST 28.16.31.10 EQ 28 .	Protocol = TCP? match -> Dest. Addr. = 28.16.31.10 match -> Dest. Port = 28 match -> PERMIT PACKET miss v DEFAULT DENY miss v DEFAULT DENY miss v DEFAULT DENY

Example Showing the Addition of a Binary Comparison Tree Constructed for the First in Sequence Rule In ACL Rule Set Into The Hash Table

Select bit string constructed from **first** ACL rule in Rule Set, utilizing the contents of those bit positions (1, 3, 4, and 34) pointed at by the Hash-Table-Balancing Bit Selection Vector, enter hash table at entry corresponding to the bits at bit positions serving as hash key index (e.g., bit position 1 contains "0"; bit position 3 contains "0"; bit position 4 contains "0"; and bit position 34 contains "0") and build binary Comparison Tree indicative of this first selected ACL rule	0000	Protocol = TCP? match -> Dest. Addr. = 28.16.31.10 match -> Dest. Port = 28 match -> PERMIT PACKET miss v DEFAULT DENY miss v DEFAULT DENY miss v DEFAULT DENY
	0001	
	0010	
	0011	
	0100	
	0101	
	0110	
	0111	
	1000	
	1001	
	1010	
	1011	
	1100	
	1101	
	1110	

577321 v3

FIG. 7D1

	1111	

FIG. 7D2

Example Showing the Construction of Balanced Hash Table Of ACL Binary Comparison Trees (cont.)

Example showing the creation of a Binary Comparison Tree for Second In Sequence Rule in Rule Set

DENY TCP ANY HOST 28.16.31.10 GT 23	Protocol = TCP? match -> Dest. Addr. = 28.16.31.10 match -> Dest. Port > 23 match -> DENY PACKET | miss v DEFAULT DENY | miss v DEFAULT > DENY

Example Showing the Addition of a Binary Comparison Tree Constructed for the Second In Sequence Rule In ACL Rule set Into The Hash Table

Select bit string constructed from **second** ACL rule in Rule Set, utilizing the contents of those bit positions (1, 3, 4, and 34) pointed at by the Hash-Table-Balancing Bit Selection Vector, enter hash table at entry corresponding to the bits at bit positions serving as hash key index (e.g., bit position 1 contains "0"; bit position 3 contains "0"; bit position 4 contains "0"; and bit position 34 contains "0") and build binary Comparison Tree indicative of this second selected ACL rule, building on any tree that may already be present for the hash table index.	0000	Protocol = TCP? match -> Dest. Addr. | miss v DEFAULT DENY = 28.16.31.10 match -> Dest. Port | miss v DEFAULT DENY = 28 match -> PERMIT PACKET | miss v Dest. Port > 23 match -> DENY PACKET | miss v DEFAULT DENY
	0001	
	0010	
	0011	
	0100	
	0101	
	0110	
	0111	
	1000	
	1001	
	1010	
	1011	
	1100	
	1101	

577321 v3

FIG. 7D3

	1110	
	1111	

FIG. 7D4

Example Showing the Construction of Balanced Hash Table Of ACL Binary Comparison Trees (cont.)

Example Showing the Creation of a Binary Comparison Tree for Third In Sequence ACL Rule in Rule Set

DENY UDP ANY HOST 30.22.21.5 EQ 11	Protocol = UDP? match -> Dest. Addr. = 30.22.21.5 match -> Dest. Port = 11 match -> DENY PACKET miss DEFAULT DENY miss DEFAULT DENY miss DEFAULT DENY

Example Showing the Addition of a Binary Comparison Tree Constructed for the Third In Sequence Rule In ACL Rule set Into The Hash Table

	0000	Protocol = TCP? match -> Dest. Addr. = 28.16.31.10 match -> Dest. Port = 28 match -> PERMIT PACKET miss DEFAULT DENY miss DEFAULT DENY miss Dest. Port > 23 match -> DENY PACKET miss DEFAULT DENY
	0001	
	0010	
	0011	
	0100	
	0101	
	0110	
	0111	
	1000	
	1001	
	1010	
	1011	

577321 v3

FIG. 7D5

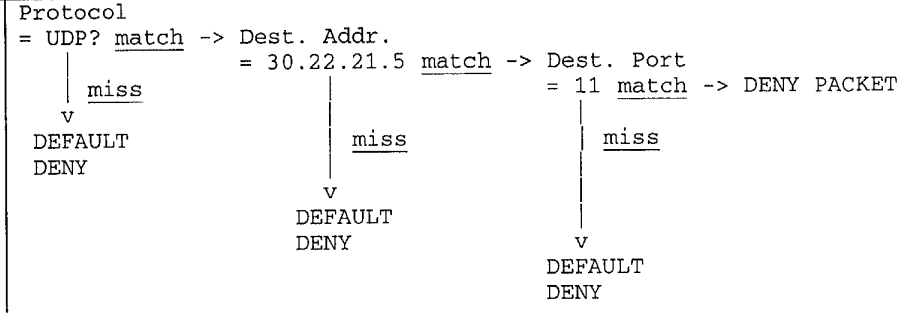
	1100	
	1101	
	1110	
Select bit string constructed from **third** ACL rule in Rule Set, utilizing the contents of those bit positions (1, 3, 4, and 34) pointed at by the Hash-Table-Balancing Bit Selection Vector, enter hash table at entry corresponding to the bits at bit positions serving as hash key index (e.g., bit position 1 contains "1"; bit position 3 contains "1"; bit position 4 contains "1"; and bit position 34 contains "1") and build binary Comparison Tree indicative of this third selected ACL rule	1111	

FIG. 7D6

Example Showing the Construction of Balanced Hash Table Of ACL Binary Comparison Trees (cont.)

Example Showing the Creation of a Binary Comparison Tree for Fourth In Sequence ACL Rule in Rule Set

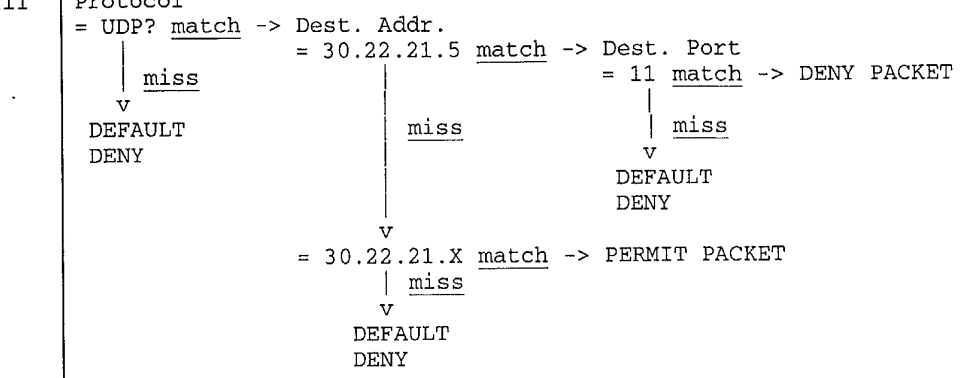
PERMIT UDP ANY HOST 30.22.21.X	Protocol = UDP? match -> Dest. Addr. = 30.22.21.X match -> PERMIT PACKETS | miss v DEFAULT DENY | miss v DEFAULT DENY

Example Showing the Addition of a Binary Comparison Tree Constructed for the Fourth in Sequence Rule In ACL Rule set Into The Hash Table

	0000	Protocol = TCP? match -> Dest. Addr. = 28.16.31.10 match -> Dest. Port = 28 match -> PERMIT PACKET | miss v DEFAULT DENY | miss v DEFAULT DENY | miss v Dest. Port > 23 match -> DENY PACKET | miss v DEFAULT DENY
	0001	
	0010	
	0011	
	0100	
	0101	
	0110	
	0111	
	1000	
	1001	
	1010	
	1011	
	1100	
	1101	
	1110	

577321 v3

FIG. 7D7

Select bit string constructed from **fourth** ACL rule in Rule Set, utilizing the contents of those bit positions (1, 3, 4, and 34) pointed at by the Hash-Table-Balancing Bit Selection Vector, enter hash table at entry corresponding to the bits at bit positions serving as hash key index (e.g., bit position 1 contains "1"; bit position 3 contains "1"; bit position 4 contains "1"; and bit position 34 contains "1") and build binary Comparison Tree indicative of this fourth selected ACL rule, building on any tree that may already be present for the hash table index	1111	

577321 v3

FIG. 7D8

Example Showing the Construction of Balanced Hash Table Of ACL Binary Comparison Trees (cont.)

Example Showing the Creation of a Binary Comparison Tree for Fifth In Sequence ACL Rule in Rule Set

DENY TCP 23.20.7.0 X.X.X.X.	Protocol = TCP? match -> Source Addr. = 23.20.7.0 match -> DENY PACKETS miss miss DEFAULT DENY DEFAULT DENY

Example Showing the Addition of a Binary Comparison Tree Constructed for the Fifth In Sequence Rule In ACL Rule set Into The Hash Table

Select bit string constructed from **fifth** ACL rule in Rule Set, utilizing the contents of those bit positions (1, 3, 4, and 34) pointed at by the Hash-Table-Balancing Bit Selection Vector, enter hash table at entry corresponding to the bits at bit positions serving as hash key index (e.g., bit position 1 contains "0"; bit position 3 contains "0"; bit position 4 contains "0"; and bit position 34 contains "X") and build binary Comparison Tree indicative of this fifth selected ACL rule, building on any tree that may already be present for the hash table index; however, since bit at bit position 34 is X, the rule will be appended at both 0000 and 0001, since bit position 34 may be either 0 or 1. In addition, since the rule itself applies to any destination address, the miss branch of all destination branches present must feed back into the source address compare instruction associated with this Fifth Rule.	0000	Protocol = TCP? match -> Dest. Addr. = 28.16.31.10 match -> Dest. Port = 28 match -> PERMIT PACKET miss miss miss DEFAULT DENY Dest. Port > 23 match -> DENY PACKET miss -> Srce. Addr. = 23.20.7.0 match -> DENY PACKET miss DEFAULT DENY -> Srce. Addr. = 23.20.7.0 match -> DENY PACKETS miss DEFAULT DENY

577321 v3

FIG. 7D9

Select bit string constructed from **fifth** ACL rule in Rule Set, utilizing the contents of those bit positions (1, 3, 4, and 34) pointed at by the Hash-Table-Balancing Bit Selection Vector, enter hash table at entry corresponding to the bits at bit positions serving as hash key index (e.g., bit position 1 contains "0"; bit position 3 contains "0"; bit position 4 contains "0"; and bit position 34 contains "X") and build binary Comparison Tree indicative of this fifth selected ACL rule, building on any tree that may already be present for the hash table index; however, since bit at bit position 34 is X, the rule will be appended at both 0000 and 0001, since bit position 34 may be either 0 or 1.	0001	Protocol = TCP? match -> Source Addr. | = 23.20.7.0 match -> DENY PACKETS | miss | v | miss DEFAULT | DENY v DEFAULT DENY
	0010	
	0011	
	0100	
	0101	
	0110	
	0111	
	1000	
	1001	
	1010	
	1011	
	1100	
	1101	
	1110	
	1111	Protocol = UDP? match -> Dest. Addr. | = 30.22.21.5 match -> Dest. Port | = 11 match -> DENY PACKET | miss | v | miss DEFAULT | DENY v = 30.22.21.X match -> PERMIT PACKET | miss v DEFAULT DENY

577321 v3

FIG. 7D10

Example Showing the Construction of Balanced Hash Table Of ACL Binary Comparison Trees (cont.)

Example Showing the Creation of a Binary Comparison Tree for Sixth In Sequence ACL Rule in Rule Set

PERMIT TCP ANY HOST 28.16.31.10	Protocol = TCP? match -> Dest. Addr. = 28.16.31.10 match -> PERMIT PACKET miss miss v DEFAULT DENY v DEFAULT DENY

Example Showing the Addition of a Binary Comparison Tree Constructed for the Sixth In Sequence Rule In ACL Rule set Into The Hash Table

Select bit string constructed from **sixth** ACL rule in Rule Set, utilizing the contents of those bit positions (1, 3, 4, and 34) pointed at by the Hash-Table-Balancing Bit Selection Vector, enter hash table at entry corresponding to the bits at bit positions serving as hash key index (e.g., bit position 1 contains "0"; bit position 3 contains "0"; bit position 4 contains "0"; and bit position 34 contains "0") and build binary Comparison Tree indicative of this sixth selected ACL rule, building on any tree that may already be present for the hash table index.	0000	Protocol = TCP? match -> Dest. Addr. = 28.16.31.10 match -> Dest. Port = 28 match -> PERMIT PACKET miss miss miss v DEFAULT DENY Dest. Port > 23 match -> DENY PACKET miss -> Srce. Addr. = 23.20.7.0 match -> DENY miss PACKET PERMIT PACKET -> Srce. Addr. = 23.20.7.0 match -> DENY PACKETS miss DEFAULT DENY

577321 v3

FIG. 7D11

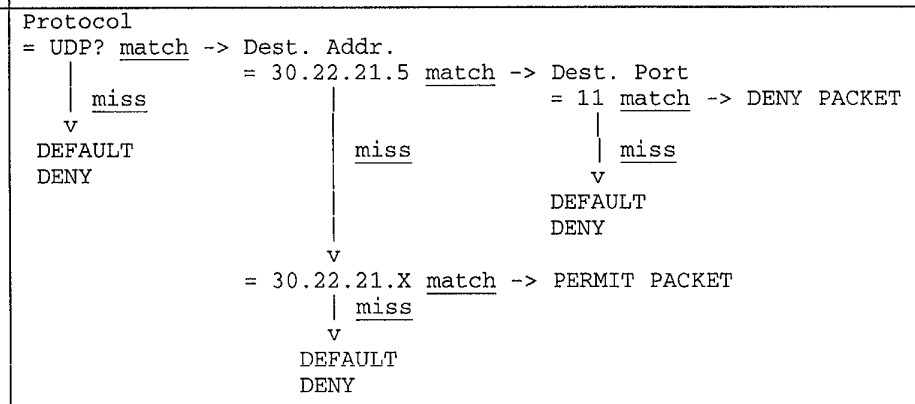
	0001	
	0010	
	0011	
	0100	
	0101	
	0110	
	0111	
	1000	
	1001	
	1010	
	1011	
	1100	
	1101	
	1110	
	1111	

FIG. 7D12

DECLARATION FOR PATENT APPLICATION AND POWER OF ATTORNEY

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below adjacent to my name.

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of subject matter (process, machine, manufacture, or composition of matter, or an improvement thereof) which is claimed and for which a patent is sought by way of the application entitled

Implementing Access Control Lists Using A Balanced Hash Table Of Access Control List Binary Comparison Trees

which (check) ☒ is attached hereto.
☐ and is amended by the Preliminary Amendment attached hereto.
☐ was filed on _____ as Application Serial No. _____
☐ and was amended on ___ (if applicable).

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information, which is material to patentability as defined in Title 37, Code of Federal Regulations, § 1.56.

I hereby claim foreign priority benefits under Title 35, United States Code, § 119(a)-(d) of any foreign application(s) for patent or inventor's certificate or any PCT international application(s) designating at least one country other than the United States of America listed below and have also identified below any foreign application(s) for patent or inventor's certificate or any PCT international application(s) designating at least one country other than the United States of America filed by me on the same subject matter having a filing date before that of the application(s) of which priority is claimed:

Prior Foreign Application(s)			Priority Claimed	
Number	Country	Day/Month/Year Filed	Yes	No
N/A			☐	☐

I hereby claim the benefit under Title 35, United States Code, § 119(e) of any United States provisional application(s) listed below:

Provisional Application Number	Filing Date
N/A	

I hereby claim the benefit under Title 35, United States Code, § 120 of any United States application(s) or PCT international application(s) designating the United States of America listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior application(s) in the manner provided by the first paragraph of Title 35, United States Code, § 112, I acknowledge the duty to disclose information, which is material to patentability as defined in Title 37, Code of Federal Regulations, § 1.56, which became available between the filing date of the prior application(s) and the national or PCT international filing date of this application:

Application Serial No.	Filing Date	Status (patented, pending, abandoned)
N/A		

I hereby appoint the following attorney(s) and/or agent(s) to prosecute this application and to transact all business in the United States Patent and Trademark Office connected therewith:

Alan H. MacPherson (24,423); Brian D. Ogonowsky (31,988); David W. Heid (25,875); Norman R. Klivans (33,003); Edward C. Kwok (33,938); David E. Steuber (25,557); Michael Shenker (34,250); Stephen A. Terrile (32,946); Peter H. Kang (40,350); Ronald J. Meetin (29,089); Ken John Koestner (33,004); Omkar K. Suryadevara (36,320); David T. Millers (37,396); Kent B. Chambers (38,839); Michael P. Adams (34,763); Robert B. Morrill (43,817); Michael J. Halbert (40,633); Gary J. Edwards (41,008); James E. Parsons (34,691); Daniel P. Stewart (41,332); Philip W. Woo (39,880); John T. Winburn (26,822); Tom Chen (42,406); Fabio E. Marino (43,339); William W. Holloway (26,182); Don C. Lawrence (31,975); Marc R. Ascolese (42,268); Carmen C. Cook (42,433); David G. Dolezal (41,711); Roberta P. Saxon (43,087); Mary Jo Bertani (42,321); Dale R. Cook (42,434); Sam G. Campbell (42,381); Matthew J. Brigham (44,047); Hugh H. Matsubayashi (43,779); Margaret M. Kelton (44,182); Joseph T. VanLeeuwen (44,383); Patrick D. Benedicto (40,909); T.J. Singh (39,535); Shireen Irani Bacon (40,494); Rory G. Bens (44,028); George Wolken, Jr. (30,441); John A. Odozynski (28,769); Cameron K. Kerrigan (44,826); Barmak S. Sani (45,068); Kenneth C. Brooks (38,393); Paul E. Lewkowicz (Reg. No. 44,870); and Theodore P. Lopez (44,881).

Please address all correspondence and telephone calls to:

Dale R. Cook
Attorney for Applicant(s)
SKJERVEN, MORRILL, MacPHERSON, FRANKLIN & FRIEL LLP
25 Metro Drive, Suite 700
San Jose, California 95110-1349

Telephone: 408-453-9200
Facsimile: 408-453-7979

I declare that all statements made herein of my own knowledge are true, all statements made herein on information and belief are believed to be true, and all statements made herein are made with the knowledge that whoever, in any matter within the jurisdiction of the Patent and Trademark Office, knowingly and willfully falsifies, conceals, or covers up by any trick, scheme, or device a material fact, or makes any false, fictitious or fraudulent statements or representations, or makes or uses any false writing or document knowing the same to contain any false, fictitious or fraudulent statement or entry, shall be subject to the penalties including fine or imprisonment or both as set forth under 18 U.S.C. 1001, and that violations of this paragraph may jeopardize the validity of the application or this document, or the validity or enforceability of any patent, trademark registration, or certificate resulting therefrom.

Full name of sole (or first joint) inventor: Haq, Faisal

Inventor's Signature: _____ Date: 1/13/00

Residence: Sunnyvale, California

Post Office Address: 1072 Durham Court Citizenship: United States
Sunnyvale, California 94087

Full name of second inventor: Lalgudi, Hari K.

Inventor's Signature: _____ Date: 1/13/00

Residence: San Jose, California

Post Office Address: 1276 Shanghai Court Citizenship: India
San Jose, California

APPENDIX

Document Number	ENG-1234
Revision	0.1
Author	Hari Lalgudi
Project Manager	Steve Kufer

Salsa4 ACL

Software Unit Design Specification

Project Headline

This document describes the software design for the input ACL processing with Salsa4 ASIC used in the TTM48 engine based linecards e.g. Gigabit Ethernet, Fast Ethernet.

Reviewers

Department	Name
Development Engineering	Steve Kufer, Manager, Software Development
Development Test Engineering	
Impacted Groups	parser-police

Modification History

Rev	Date	Originator	Comment
0.1	07/15/99	Hari Lalgudi	Initial Release

Definitions

This section defines words, acronyms, and actions which may not be readily understood.

SALSA4	ASIC on the GSR TTM48 Linecards that support ACL lookup in hardware Refer to ENG-35918 for complete description. In this document Salsa by default will refer to Salsa4 only.
ACL	Access control lists used for packet classification and filtering. Each ACL has number of items, each specifying permit/deny action for packets matching the source, destination
Ex-ACL	Extended ACL, which matches more fields in IP header than standard ACL (source IP address). Fields include Destination IP address, source and destination TCP/UDP ports, protocol type.
TTM48	Time to Market OC48 engine and line cards built on this Engine(also refered as TTM)

GE	Gigabit Ethernet GSR linecard with TTM48 engine and Salsa4 ASIC (unless otherwise specified)
FE	8 port Fast Ethernet GSR linecard with TTM48 engine and Salsa4 ASIC (unless otherwise specified)
S-ACL	ACL lookup done by Salsa4
FIB, CEF	Forwarding Information Base or Cisco Express Forwarding, which is an mtrie used for lookup(based on destination IP address) to forward packets
CAR	Committed Access Rate. It is an IOS feature that may use ACLs to match packets and verify conform/exceed rate limits and perform actions such as permitting, dropping or marking the packet.

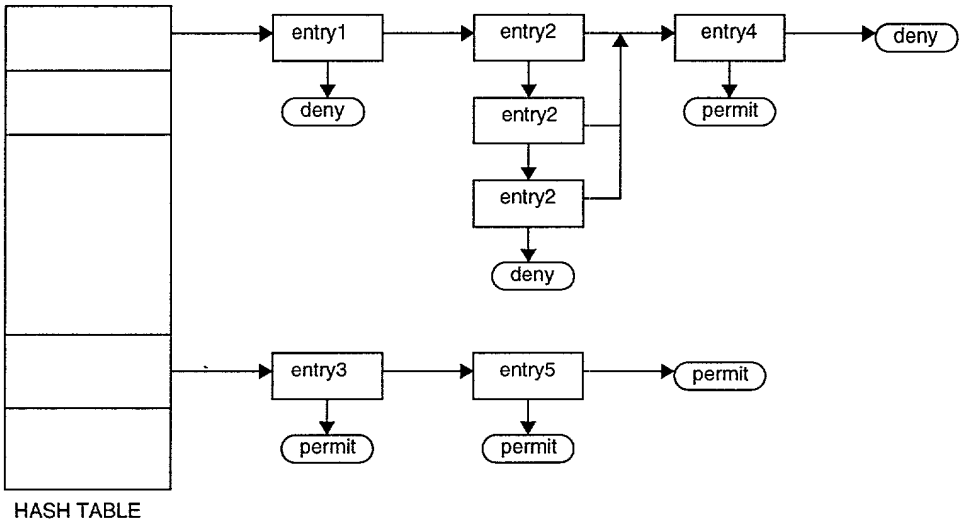
1.0 Problem Definition

TTM48 linecards without Salsa4 ACL hardware support, do extended ACL lookup in software. This CPU intensive operation results in a performance degradation from linerate (650 Kpps) without ACLs to 32Kpps with 512 ACL items. With Turbo or compiled ACLs in software, performance drop is from 650Kpps to a constant 270 Kpps (independant of the number of ACL items).

TTM ACL hardware solution needs to (a) presetve the order of items and checks in ACLs (b) provide full filtering or classification functionality(e.g. source and destination IP address, TOS, TCP/UDP port ranges) currently supported in software (c) higher performance than Turbo ACLs (270Kpps) for 10-1000 ACL items..

Salsa4(ENG-31157) solution is a hardware engine that traverses a hashed linked list of nodes(figure below), that is equivalent to the extended ACL, to permit or deny the packet. Each node(64 bits) is an instruction to Salsa4 to compare one or more fields(source/destination IP address, TCP/UDP port) in the incoming packet against an ACL parameter, and find the next match or miss nodes. The last(Stop) node specifies permit/deny value and more information for software to execute any other actions(e.g. ACL accounting, ACL logging or TCP flags checks).

f



Salsa4 ACL support is only for input ACLs, since Salsa4 does not do ACL lookup after FIB lookup is done when output adjacency/port areidentified. Also, Salsa4 does not support subinterface ACL, since subinterface identification (e.g. based on VLAN) is difficult in the given hardware.

2.0 Design Considerations

S-ACL(Salsa4 ACL) software tasks are:

1. Parse IOS ACL configuration to create the hash table and list of nodes.

2. Switching path code to process S-ACL node information(e.g. permit/deny, ACL logging, check TCP flags)

3. Handle error conditions(e.g. maximum number of nodes read) and continue software lookup on the ACL hash list in selected cases.

4. Provide user configurations to enable S-ACLs and to do performance tuning by weighting selection of hash bits.

2.1 Requirements/Constraints on parsing IOS ACL to Hash Table and Nodes

S-ACL nodes should preserve order of ACL item checks. S-ACL performance crucially depends on the average number of nodes Salsa4 processes before coming to the Stop node. This depends on

(a) the bits selected for hashing

(b) the parsing algorithm for creating nodes from ACLs.

(c) traffic pattern to different Stop nodes.

2.2 Constraints on Switching Code for S-ACL support

1. There should be no performance impact on GSR linecards that do not use Salsa4 ASIC.

2. TTM48 linecards can achieve 650Kpps by offloading FIB lookup in Salsa and the software saving cycles by not caching and invalidating packet headers in the L2 cache. To achieve performance better than 270Kpps(Turbo-ACLs), software should only cache/invalidate packet headers in L2 cache only (a) if Salsa4 does not find a valid(permit/deny) Stop node or (b) need to apply other features(input CAR/accounting).

2.3 S-ACL vs Turbo-ACL

Although Turbo-ACL gives 270Kpps on GSR LC with input ACLs irrespective of the number of ACLs, , the performance drop from 650 Kpps for TTM48 LCs is significant. S-ACL performance is dependant on the number of ACL items (which determines number of ACL nodes). We do not provide automatic selection of Turbo-ACL or S-ACLs. Here are the guidelines for the user for manual selection via configuration:.

(a) For output ACLs, use Turbo-ACLs

(b) For input ACLs where number of input items leads to performance < 270Kpps, use Turbo-ACLs, otherwise use S-ACLs.

2.4 Separation of hardware dependant and hardware independant modules

S-ACL hash tables and trees should be made hardware independant so that pure software lookups without hardware assist is possible This is useful for (a) linecards without Salsa4 and (b) ACL lookup for packet classification purposes other than filtering (e.g. ACL based CAR)

2.5 Memory requirements for S-ACL

The Match/Miss address in each ACL node is derived from the "node base"(10b) and match/miss "offset"(8b). The algorithm to create ACL nodes needs to make sure that the "node base" is the same for the match and miss nodes. We guarantee this by allocating 2^18 nodes for each S-ACL node memory.

Thus the memory requirements for S-ACL on 8 port FE LC is (8 * 2^18 * 8B) = 8MB

3.0 Functional Structure

The S-ACL software is separated into Salsa4 specific module and hardware independant ACL Hash Table module.

The Salsa4 module sets up Salsa4 registers to enable/disable ACL lookup, setup hash table addresses, select hash tables based on multi ported linecards (e.g. FE), and to keep history of errors during S-ACL lookup.

The ACL Hash Table module however needs parameters from the hardware dependant module to allocate memory that is consistent with hardware requirements (e.g. 8B aligned or hash table next to node memory map)

The current scheme of sending ACL config is sent from RP to LC via FIB IPC messages is unchanged. While parsing interface ACL config on LC, if S-ACL interface is affected, then we do the following:

(1) disable Salsa4 from using S-ACL hash tables.

(2) clear the current S-ACL hash table and node memory map.

(3) recalculate hash key and insert hash nodes for each ACL item. For multiported linecards(FE) since there is only one hash key for all hash tables, this implies recalculating hash key based on all FE interface S-ACL configurations and inserting ACL nodes for all FE interface ACL items.

4.0 Packet Flow

We need to add support for S-ACL in the software switching code without affecting performance of existing linecards that do not use Salsa4. Switching vectors introduced by Netflow on GSR(12.0(6)S) allows us to add new features in the switching path without incurring the runtime performance. Currently, there are 4 switching vectors for TTM path: no flow/no feature, flow/no feature, no flow/feature, flow and feature (feature refers to FIB/CEF features such as input/output ACLs or CAR or accounting).

S-ACL support makes this 8 switching vectors: no S-ACL/no flow/no feature, no S-ACL/flow/no feature, no S-ACL/no flow/feature, no S-ACL/flow/feature, S-ACL/no flow/no feature, S-ACL/flow/no feature, S-ACL/no flow/feature, S-ACL/flow/feature. We select S-ACL switching vectors if we find Salsa-4 version (Salsa L3 Asic Id Register >= 4).

In S-ACL switching vectors, if Salsa4 gets a valid FIB leaf, we read the S-ACL registers (while still working in L2 uncached mode for accessing bhdr and IP ptrs). If there are no S-ACL errors (S-ACL status register), and no other feature checks (e.g. input/output CAR/accounting) need to be done, we switch the packet in uncached mode. Else, we cache the packet, check other features, switch the packet and invalidate the L2 cache (and incurring the performance penalty of dropping to 300-400Kpps).

If S-ACL is successful, we need to disable input ACL checks in the FIB feature path. This is done by setting the platform specific FIB control block parameter to skip input ACL checks.

5.0 Salsa4 error handling

Salsa4 may report the following types of errors in S-ACL processing:

- ACL Engine Maxed out error: Salsa4 did not reach a STOP node(permit/deny) at the end of maximum number of lookups (software configurable register). The DRAM Catastrophic Error Address register gives the address information.
- ACL Engine uncorrected ECC error: Salsa4 encountered a uncorrectable ECC error during a DRAM read. The DRAM Catastrophic Error Address register gives the address information.
- ACL Engine out-of -range error:

S-ACL hash table errors(due to software bugs or insufficient memory) are reported by errmsgs. The following debug commands on the linecard allows further debugging:

- debug ip access hash
- debug ip access detail-hash

The following commands show the S-ACL hash table and nodes information. If the above debugs are turned off, only the "useful" nodes are display, otherwise all nodes in the hash table, including stop nodes are printed.

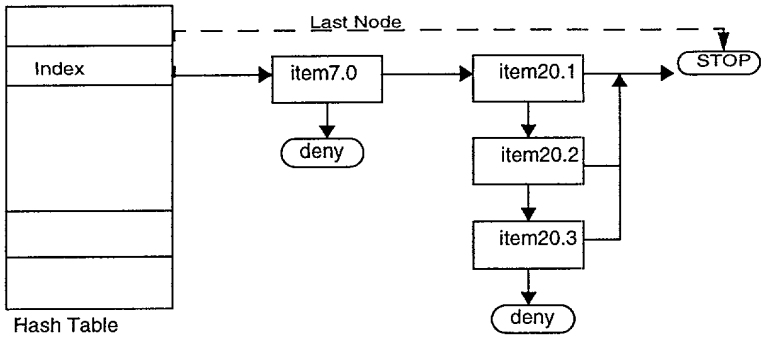
- show access hash <port-number> {nodes}

6.0 Algorithmic Description

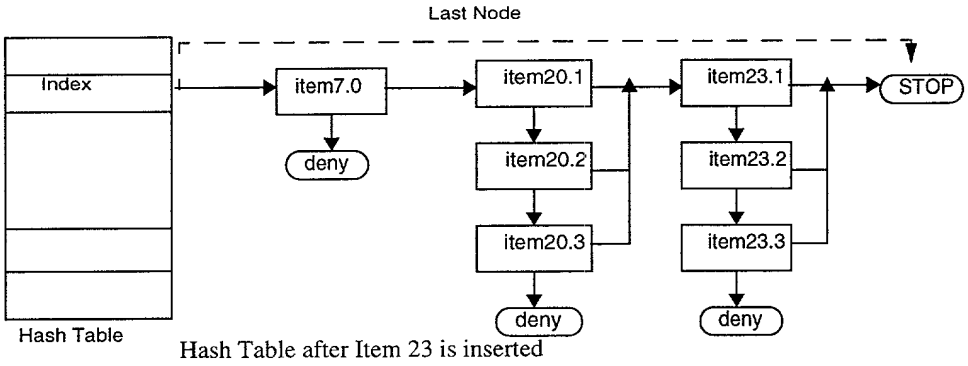
6.1 ACL Items mapping to ACL Nodes

Each ACL item may contain checks(exact match or range) for source IP address, destination IP address, protocol type, source and destination TCP/UDP port numbers and TOS. This translates to multiple ACL nodes, each node checking one value of the field (or multiple fields for TOS/Protocol). When we parse each item we set the "Miss" address in each of these nodes to a new Stop node, which is that index's "last node". Since we need to keep the ordering among ACL items, we use this "last node" as the starting node for the nextACL item that hashes into the same hash index

For example, item 7and 20 of an ACL, which map to the same hash index, are inserted in the table with a last stop node. Now we want to insert item 23, also hashing to the same index, starting from the last node.



Hash Table before Item 23 is inserted



Hash Table after Item 23 is inserted

6.2 Dynamic ACL Hash Key selection algorithm

Using the above ACL item to node allocation method, the worst performance is obtained when Salsa4 is made to walk all the ACL nodes for given hash index and terminates at the last node

We can reduce the distance to the last node by picking the right bits for ACL Hash Key selection.

Consider each ACL item to be represented as a "select" bitmask of <src_ip, dst_ip, protocol, src_port, dst_port, tos>, where a field's bit = 1 if the ACL item checks the field, else field's bit = 0.

If a field (e.g. bit 9 of destination IP address) is selected for determining the hash key, and if the corresponding bit in the item's "select" bitmask is 1, then the item's ACL nodes need to be inserted only in the hash index for which the bit is either 0 or 1 (depending on ACL item's value). If the "select" bitmask is 0, then the item's ACL nodes need to be inserted in all hash indexes where this bit is either 0 or 1.

Our algorithm maintains two counts, zero_count(for bit=0) and one_count(for bit=1), for each bit in the "select" bitmask. For each ACL item, we get a "per-item-wt" by multiplying the number of nodes for that ACL item with a per-ACL weight(default 1). If the "select" bitmask for that bit = 0, we add the "per-item-wt" to both zero_count and one_count. Else, if the item bit value in item is 0, we add the "per-item-wt" to zero_count only. Else if the item bit value is 1, we add "per-item-wt" to only one_count.

Worst case performance penalty of not selecting a bit in the hash key is given by max(zero_count, one_count) for that bit. Worst case performance of selecting the bit in the hash key is given by the min(zero_count, one_count) for that bit.

We seek to minimize the maximum performance penalty of selecting the bits. Hence we sort the min(zero_count, one_count) of all the bits and pick up the first "n" (for Salsa4, n = 10) bits. If there are two bits with the same min() value, the max() is used to differentiate the bits.

The per-ACL weights represents the amount of traffic hitting each ACL. These weights may be assigned by the user or calculated by the amount of traffic hitting the ACL as a percentage of all the traffic.

Traffic distribution may be skewed towards one protocol (e.g. HTTP with TCP/IP) and the ACLs may not represent this factor. We use a "per-field" weight to compare the performance penalty of selecting a field that may not be varying as much as other field.

7.0 SW Restrictions and Configuration

(1) S-ACL does not currently support ACL lookup for CAR or output ACLs.

(2) Currently support is only in 12.0S (no support is planned for 11.2GS releases)

(3) S-ACL is not supported for subinterface ACLs.

(4) S-ACL tree updates are not done on the fly, since Salsa4 may walk down old/unused paths.

8.0 HW Restrictions and Configuration

(1) For multiported TTM LCs (e.g. FE), we do not have multiple hash keys for the different hash tables. Since same hash key is used for multiple ports, a long ACL on one port may influence selection of hash key bits, thus influencing performance on other FE port using same hash key.

(2) Port information for selecting hash table is either in the INPUT_INFO field of BHDR or the 4B POS header(FE).

(3) Maximum lookup counter is implemented to prevent Salsa4 from getting lost in a corrupt ACL tree. This requires software to perform the rest of the lookups if a valid result requires more S-ACL checks.

(4) 8 port FE LC requires 4MB memory (8 chunks aligned at 256KB) for the 8 hardware ACL hash tables. If this memory is not available during initialization, S-ACL support may fail.

9.0 External Restrictions and Configuration

S-ACL support for GE is dependant on GSR product team approval for Salsa4 GE LC as an "enhanced" linecard or a new feature on existing linecards that may require a hardware upgrade of existing linecards.

10.0 Development Unit Testing

S-ACL unit test cases should verify the functionality for different types of ACL nodes(permit and deny) that can be generated by ACL configurations:

1. Source IP address with mask of 0.255.255.255, 0.0.255.255, 0.0.0.255 and 0.0.0.0

2. DestinationIP address with mask of 0.255.255.255, 0.0.255.255, 0.0.0.255 and 0.0.0.0

3. Protocol types: TCP, UDP, ICMP, IGMP, IGRP, EIGRP, IGRP, IPINIP, NOS, OSPF

4. TCP/UDP Source/Destination Port: port numbers with operations eq, lt, gt.

5. ICMP/IGMP Source/Destination port numbers with operations eq, lt, gt.

6. TOS values with combination of Source/Destination port numbers indicated above.

Other unit tests are to populate the ACL Hash table with at least one ACL item and check for validity. Example:

(a) Configure 1024 ACL items each with incrementing source/destination IP address, with alternating permit/deny values in each item.

(b) Send traffic for 1024 source/destination IP address and verify that Salsa drops/permits the packet and updates the right ACL item counters.

Other unit test is use "S-ACL debug" mode, where we configure 1000+ item list with incrementing source/ destination IP address or TCP ports or protocol fields. Configure "debug access switch" on the linecard that verifies the output of the S-ACL lookup with the conventional or Turbo-ACL lookup. Scripts then send traffic with incrementing source/destination IP addr or port or protocol numbers. Any mismatch between S-ACL and Turbo-ACL is reported via error msgs with packet dump and Salsa4 registers for further debugging.

References

Salsa4 ASIC Hardware Specification, ENG-31157, Faisal Haq

Appendixes

Specification Review

cisco Systems

Document Number	ENG-31157
Revision	0.2
Author	Faisal Haq
Project Manager	Kamal Avlani

Salsa4 ASIC

Hardware Specification

Project Headline

Summary of modifications in revision 4 of the Salsa switching ASIC

Approvals

Name	Approval Date
Faisal Haq HW Design Engineer	-
Kamal Avlani Program Manager	
Brad Wurtz HW Design Engineering Manager	-

Modification History

Rev	Date	Originator	Comment

Table of Contents

Section 1. Introduction

As the Salsa ASIC is being employed in multiple GSR 12000 Linecards, new requirements for this design have emerged that necessitate a spin. The following functionality is being considered for the spin, which is being called SALSA4:

- ECC Protection on Routing Table Memory (EDO) Interface
- Hardware based access list implementation.

Each of these features are discussed in much detail in the following sections. They have been implemented with the following assumptions:

- Pin-for-pin compatibility with previous revisions of Salsa. This limits Salsa4 to the original 503-pin EPBGA package and the LSI G10p process.
- Quick turnaround solution, that provides chips in hand within 3 months.
- Low risk changes that can be easily synthesized and verified within the allotted turnaround time.

Section 2 Functional Description of Modifications

2.1 Hardware-based access-list filtering

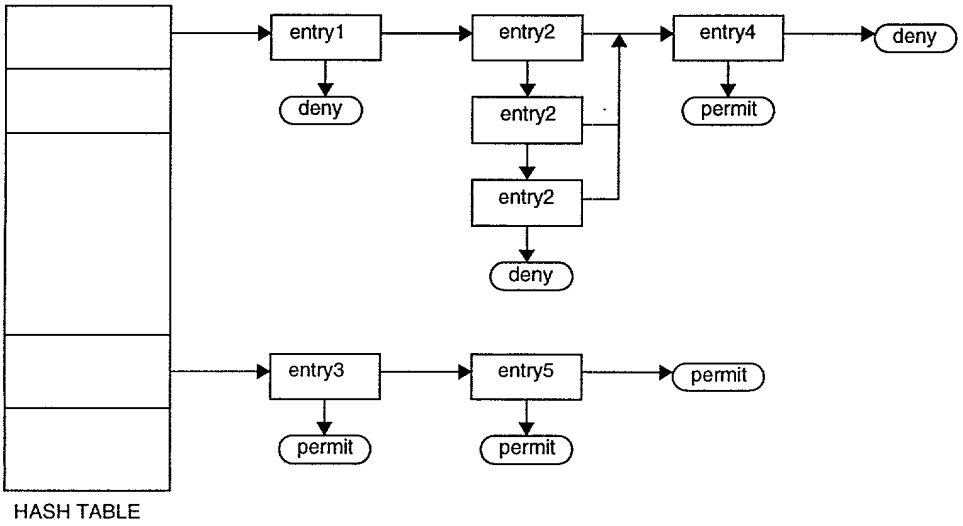
TTM based multi-port linecards (primarily DS3, Fast Ethernet and Gigabit etheret) are not expected to sustain an acceptable packet rate with software-only based extended access lists. The following performance has been demonstrated with "benchmark" access-lists obtained from ISPs.

Table 1: Performance of Extended Access Lists on exisitng GSR linecards, using a software-only approach

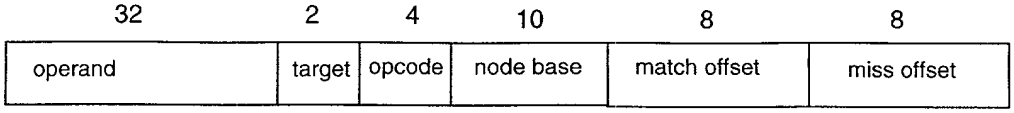
Access List	Linecard	L2 Cache?	Performance
Internal (256 entries)	Jaguar	Yes	38Kpps
Internal (512 entries)	Jaguar	Yes	32Kpps
Internal (256 entries)	QOC3 POS	No	35Kpps

2.1.1 The ACL Tree

Salsa4's solution is a hardware engine that traverses a linked-list implementation of the ACL (the ACL tree). There are several start points to the tree, and these are stored in a hash table that is accessed once per packet. There is one tree for each port of the linecard. Each entry in the user's access-list corresponds to one or more nodes in the ACLtree. The diagram below shows the implementaion:



Each node in the tree is an instruction to Salsa4, to compare one or more fields of the incoming packet, against a value determined by the ACL. The node is a 64bit entity of the following format:



operand — This field is the value to be compared against, but its format depends on the type of comparison defined by the opcode bits. For a 32 bit compare, this field is the entire 32bit value to be compared against. For 16 bit compares, the upper 16bits is the value and the lower 16bits are mask bits.

target — The target field indicates to Salsa4, which field in the incoming packet is to be used for the compare. The decode is:

```
00    Source Address
01    Destination Address
10    The 32bit concatenation of {Dest.Port, Protocol,TOS}
11    The 32bit concatenation of {SourcePort, Protocol,TOS}
```

opcode — An opcode to define the type of compare:

```
0000  STOP. (See operand field for information on permit/deny)
0010  16bit compare of UPPER 2 BYTES of target
0011  16bit compare of LOWER 2 BYTES of target
0100  16bit greater-than compare of UPPER 2 BYTES of target
0101  16bit greater-than compare of LOWER 2 BYTES of target
0110  32bit compare against ACL Mask Register 0
0111  32bit compare against ACL Mask Register 1
1000  32bit compare against ACL Mask Register 2
1001  32bit compare against ACL Mask Register 3
1010  32bit compare against ACL Mask Register 4
1011  32bit compare against ACL Mask Register 5
1100  32bit compare against ACL Mask Register 6
1101  32bit compare against ACL Mask Register 7
1110  32bit compare against ACL Mask Register 8
1111  32bit compare against ACL Mask Register 9
```

node base — This is the 10bit base address for the next node addresses. This base is not to be confused with the PORT base address, discussed later.

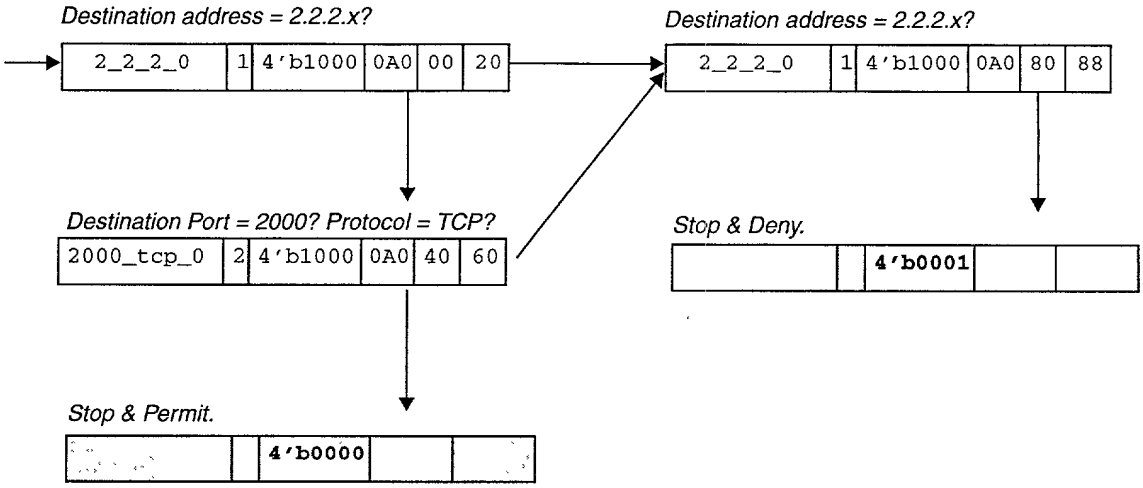
match address — Base offset for the next address, if the operand matches the target

miss address — Base offset for the next address, if the operand does NOT match the target

Thus, as an example the access-list entries..

```
access-list 101 permit tcp any 2.2.2.0  0.0.0.255  eq  2000
access-list 101 deny tcp any 2.2.2.0  0.0.0.255
```

would be implemented in the following manner, in the ACL tree:



Because of the inter-dependance of entries in the list, and as the example above also reveals, it is very important for software to maintain the user's order of ACL entries when constructing the tree. Without this, it is possible for packets to be permitted or denied incorrectly. It is suggested that software start from the bottom of the list and proceed upwards, inserting entries to the left of nodes in the tree,

2.1.2 A Perl program to determine Hash size

A perl script has been written to estimate the performance in pps, of access list lookups after they are hashed. The program converts actual ACL configurations into a hashed array of compare instructions, and provides relevant statistics on the size of trees stemming from the hash table buckets. The program has been written by Faisal Haq and Hari Lalgudi, and is included in an Appendex at the end of this document.

Based on a large database of customer access-lists (used for the design of the Toaster ASIC), and on ACL configurations sent to the Optical Internetworking BU from AOL, results from this program suggest a 1024-bucket hash table as being most optimal. Larger hash tables tend to reduce the overall depth of trees stemming from the buckets. Smaller hash tables consume less memory and are quicker to update. It was found that the incremental reduction in average tree depth was lowest in going from 1024 to 4096-bucket hash tables. Thus we havev chosen 1024 as the size of the hash table, implying a 10-bit hash key.

2.1.3 Determining which fields are used in Hash Key

Many of the entries in a list are mutually exclusive and it is this characteristic that allows for a performance improvement through the use of a hash table. For instance, we could group all entries in a list by the first ten bits of the destination address (still preserving order within groups!!). Then for each packet, Salsa4 would use the first ten bits of the destination address to lookup a 1024-deep hash table that would then point it to all related ACL entries.

Grouping by destination address as it turns out may not be optimal for two reasons. 1) Entries with an "any" in the destination address field would have to be replicated on all 1024 buckets, which would make the table large and impact performance for every packet being hashed. 2) Depending on where it is deployed, the linecard may receive packets destined for only a few addresses, and Salsa4 would not use the hash table uniformly over all buckets.

After many iterations with various keys, it was determined that no one key would prove optimal across the entrie suite of customer ACLs. It was then decided to go with a software-selcectable key that was customized for the ACL entered by the customer. The criteria for selection is for bits that occurr most frequently in the ACLs and occur evenly as a 1 and a 0.

So in the example below,

```
access-list 101 permit tcp any 2.2.2.1
access-list 101 deny   tcp any 2.2.2.254
```

the last 8 bits of the destination address would be among the bits chosen for the key since they occur in both entries, once with a value of 0 and an once with a value of 1. In contrast, the remaining bits of the destination address occur only as one value. This algorithm was writen into the perl program mentioned in Section 2.1.2.

The hardware design in Salsa4 allows software to select the fields used in the key, on a per-bit basis.

2.1.4 ACL Search Start and Stop Conditions

Once a packet is received in Salsa4, the MTRIE lookup logic will commence. It will complete within 4 DRAM lookups. Once the state-machine controlling this activity has returned to an IDLE state, the ACL lookup state machine shall begin. It generates a hash key using bits from the incoming packet, as specified by the software-programmed registers.

The hash key points Salsa to a chain of ACL nodes. An internal maximum-counter is clocked on each node lookup. Should that counter reach 0, the ACL lookup is terminated and the results are layed out in the "ACL Engine Status Register" on page 47. However, if a STOP node is found at or before then, the ACL lookup is ended and the results are presented in this register.

A STOP node has a special format, as shown below:

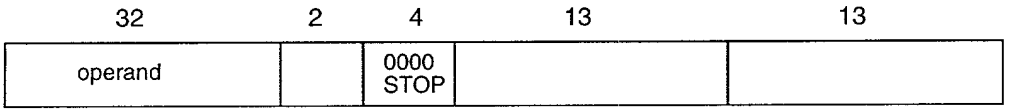


Table 2: Decode of information stored in the OPERAND field of a STOP node.

Bits	Description
[31:4]	ACL number associated with this STOP
[3]	Deny/Permit 0 - Deny 1 - Permit

Table 2: Decode of information stored in the OPERAND field of a STOP node.

Bits	Description
[2:0]	Reserved, 0's

The contents of the operand field are copied into the "ACL Engine Status Register".

2.1.5 Limitations of the Salsa4 Approach

- The hardware to lookup the hash table and then follow the ACL tree will be added as an extension of the MTRIE lookup engine on the Rx BMA interface. This produces the most significant limitation of the Salsa4 approach -- it can ONLY work on the Rx side. Output port based access-lists would still have to be done in software.
- Salsa4 must handle per port access lists. Support for upto 8 individual ports is provided.
- Port information for each packet can be found in either the INPUT_INFO field of the BHDR or in the 4-byte POS header (as used by the Eiffel project). These are the only 2 places that Salsa will look for port information.
- A maximum lookups counter will be implemented as well, to prevent Salsa4 from getting lost in a corrupted ACL tree. This programmeable counter maxes at 1024 lookups, after which the packet is presented to the processor along with the address of the most recent access. Our tests show a number of cases where the lookups exceed 32. Beyond this point it is better to have software perform the lookups, since it may have better performance than hardware, given the L2 cache onn the board.
- The node format of a node-base, with 8bit match and miss offsets means that miss/match nodes stemming from the current node, MUST be within 2K Bytes of each other.
- The ACL tree size per port is limited to 2^{11} . With 18 bits (excludes 3 LSbits, since nodes are on 8byte address boundaries) remaining for addressing, we have a per-port limitation of 256K nodes. Note: that is 256K nodes, NOT acl-entries!!. Note that each ACL entrie may be implemented in as many as 4 nodes.
- ACL entries with protocols not using source/desintation ports (icmp, igmp, eigrp, gre, igrp, ipinip, nos and ospf.) will be treated as having "don't care" ports. This is because Salsa will not recognize these types of protocols and will treat them like tcp/udp. As a result these entries will be replicated over any buckets that are represented by PROTOCOL bits in the key. (That was pretty confusing)
- Tree updates on the fly if needed, would require special handling by software to make sure Salsa4 isn't sent down old or null branches.

2.1.6 Implementation in Hardware

The implementation is best summarized in the diagrams that follow. It involves an internal state machine that iteratively walks through the ACL tree, starting first at a hash table, and subsequently following either the match or miss address of each ACL node it encounters. A maximum of 1024 ACL nodes will be iteratively read, after which the state machine stops and delivers the last read node. Software will then continue the search as needed. Ofcourse, if the state machine encounters a "Stop with Deny" or "Stop with Permit" node, that too will terminate the search.

The area for this logic is approx twice that of the MTRIE lookup engine in Salsa4. That area was approx 8000 gates. There should not be any difficulty fitting this additional logic into the existing floorplan.

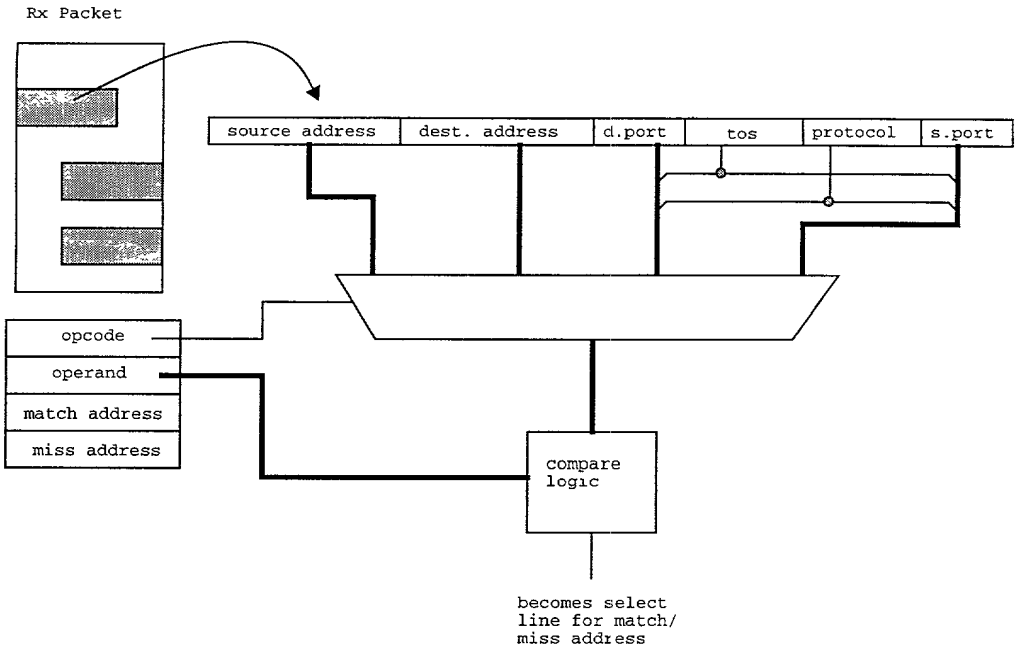


Figure 1: Comparison logic used to match/miss on a node's operand

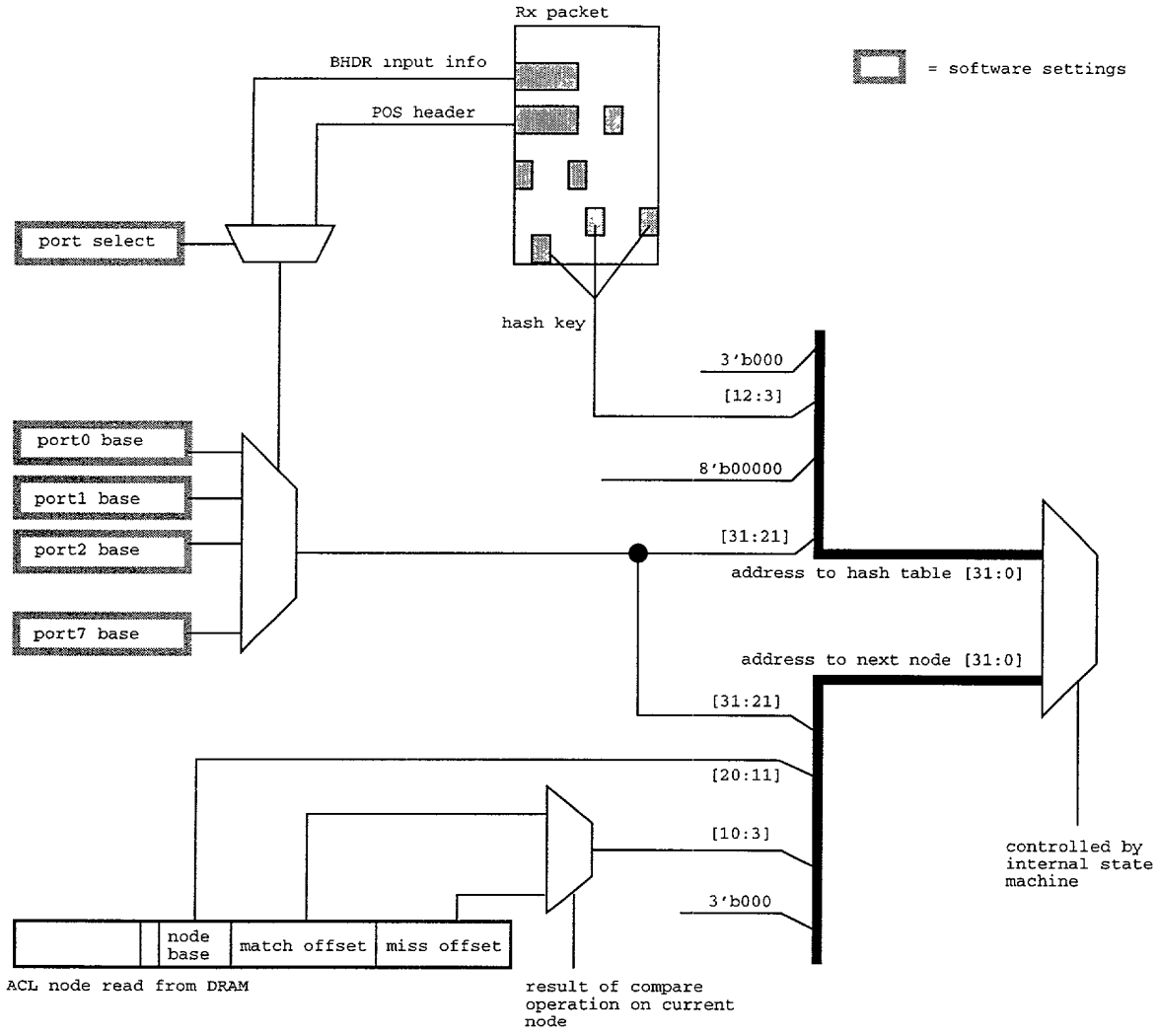


Figure 2: Addressing logic for ACL lookup engine

2.1.7 Expected performance

Performance calculations with Salsa4 are made complicated by the hash table. Some lists off the hash table may be over 70 nodes long, others just 1 node long. The worst, typical and best case numbers below use the longest, average and shortest length lists in each hash table to illustrate this fact. As discussed before, we expect incoming traffic to hit all buckets of the hash table, evenly and therefore, the typical number would be most likely seen.

These calculations are based on a 650Kpps packet rate, with any ACL engine delays added to the 1.538usec per-packet switching delay. Each "check" takes 160ns (includes RAS precharge time on the EDO) to complete.

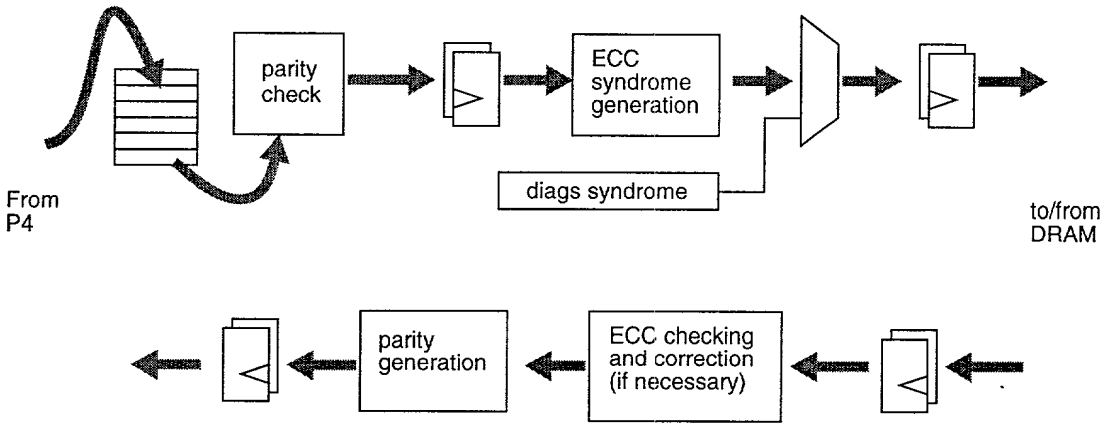
Table 3: Expected Performance of Extended Access Lists with Salsa4's ACL Engine

Access List	Performance Worst	Performance Typical	Performance Best
Internal (256 entries)	28Kpps	266Kpps	650Kpps
Internal (512 entries)	14Kpps	171Kpps	650Kpps
AOL (315 entries)	91Kpps	352Kpps	558Kpps
McGill (91entries)	86Kpps	352Kpps	485Kpps
ATT (322 entries)	192 checks 31Kpps	7 checks 376Kpps	2 checks 538Kpps
BT (1035 entries)	214 checks 28Kpps	24 checks 186Kpps	2 checks 538Kpps
SWBell1 (940 entries)	1768 checks 5Kpps	113 checks 51Kpps	4 checks 460Kpps
SWBell2 (821 entries)	488 checks 12Kpps	10 checks 319Kpps	0 checks 650Kpps

2.2 ECC Protection on DRAM interface

ECC protection on all 64bits is being added between Salsa4 and the EDO DRAM Routing table Memory. The 8 additional bits on this interface currently being used for parity, will be used as part of the ECC syndrome.

The integration of the ECC data path with the existing parity checking path is shown below:.



2.2.1 Parity handling on DRAM writes by P4

During a write, the only check is for parity on data being read out of the write buffer. If a parity error is detected,
- interrupt is sent to the processor
- write address is captured
- appropriate bits are set in the "DRAM Error Status Register"
- write is aborted.

If there is no parity error or if checking is disabled, parity bits are dropped, an ECC syndrome is calculated and written to DRAM. It is possible for software to substitute the Salsa-calculated syndrome with a value written into a register.

2.2.2 Single-bit error on a DRAM read by P4

If ECC-single-bit-error-notification is enabled,
- the processor is interrupted
- DRAM address is captured.
- Corrected data (if single-bit-error-correction is enabled) or original data (if not) is passed on to the parity generation logic and then on to the P4
- the occurrence of this error is captured in a clear-on-read bit in the "ECC Status Register"

if ECC-single-bit-error-notification is disabled:
- the processor is not interrupted
- address and status are still captured
- Corrected data (if single-bit-error-correction is enabled) or original data (if not) is passed on to the parity generation logic and then on to the P4

2.2.3 Multi-bit error on a DRAM read by P4

It is not possible to correct this type of error.

If Bus-error-on-multibit-ECC-error is enabled
- Bad data is forwarded to the P4 with good parity, but with a bus error.
- DRAM address is captured
- the occurrence of this error is captured in a clear-on-read bit in the "ECC Status Register"

If Bus-error-on-multibit-ECC-error is disabled
- Bad data is forwarded to the P4 with good parity, and no bus error
- address and status are still captured

2.2.4 Single/Multi Bit Errors on DRAM reads by ACL/MTRIE engines

When the ACL and/or MTRIE LU engines encounter an ECC error, notification is made through status registers associated with each incoming packet.

If ECC-single-bit-error-notification is enabled,
- a bit is raised in either the "ACL Engine Status Register" or the "Receive BMA Packet Synopsis Register".
- DRAM address is captured.
- Corrected data (if single-bit-error-correction is enabled) or original data (if not) is used in the lookup
- the occurrence of this error is captured in a clear-on-read bit in the "ECC Status Register"
- the engine continues searching

if ECC-single-bit-error-notification is disabled:
- ECC SBE's will not set the error bit in the engine status registers
- address and status are still captured
- Corrected data (if single-bit-error-correction is enabled) or original data (if not) is used in the lookup
- the engine continues searching

If Bus-error-on-multibit-ECC-error is enabled
- a bit is raised in either the "ACL Engine Status Register" or the "Receive BMA Packet Synopsis Register".
- DRAM address is captured
- the occurrence of this error is captured in a clear-on-read bit in the "ECC Status Register"
- the engine stops immediately

If Bus-error-on-multibit-ECC-error is disabled
- ECC MBE's will not set the error bit in the engine status registers
- address and status are still captured
- the engine stops immediately

Section 3. Manufacturing & Test Considerations

The following is the output of lsistats, on the first revision of Salsa. It is packaged in a 503 pin EPBGA.

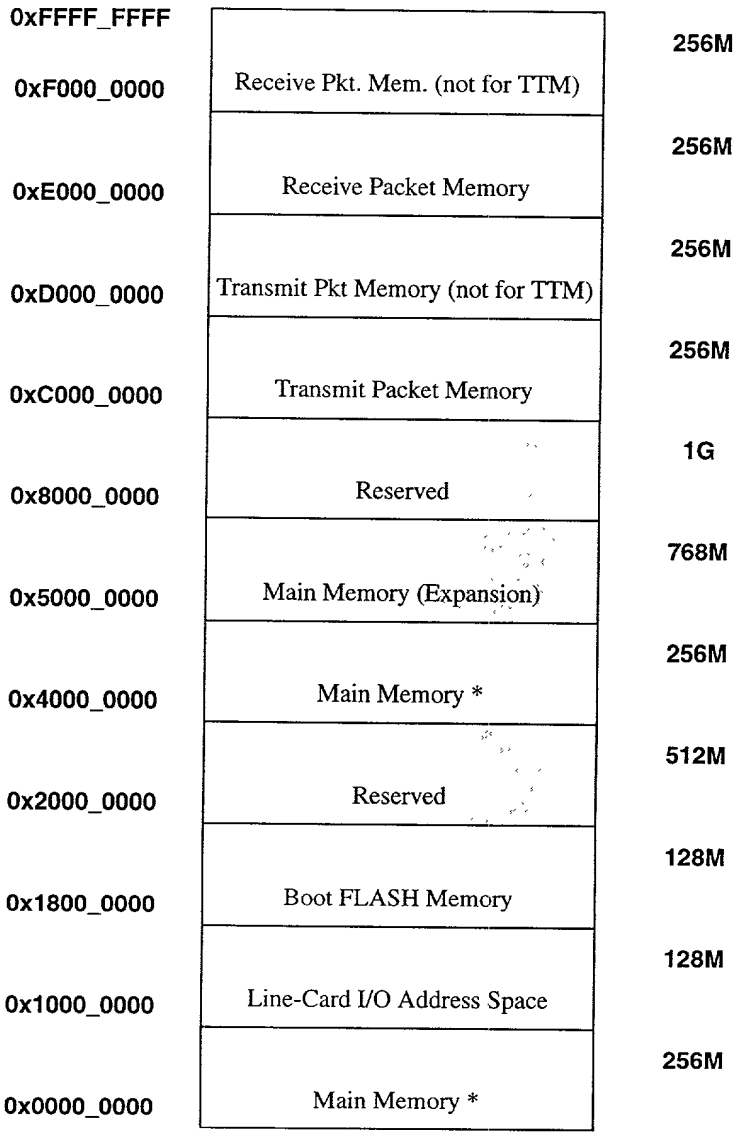
```
Section One: Design Summary
===========================

Top Module:             salsa
Technology:             lcbg10p
Array Type:             lcbg10p
Mftg Code:              Not Available.
Pad Pitch:              2.9wire
Dimension:              (X = 9.510, Y = 9.510)
Routing Layers:         2
Package Name:           iw51
Package Desc:           Not Available.
Primary Voltage:        None Specified.
Secondary Voltage:      None Specified.


Section Two: Design Statistics Summary
======================================

(1). I/O Statistics

Input Pins Used:        31      Input Pads Used:        29      Input Slots Used:       31
Output Pins Used:       67      Output Pads Used:       1       Output Slots Used:      1
Bidirect Pins Used:     242     Bidirect Pads Used:     308     Bidirect Slots Used:    308


Total Pins Used:        340
Total Pads Used:        338
Total Slots Used:       340


(2). Design Statistics

Logic Units Used (lu):          360655.00
Megacell Units Used (mu):       227083.17
Chip Raw Units:                 3249856.00
Logic Units Usage:              0.119
Chip Usage:                     0.181

Total Cells:                    26592
Total Cell Types:               163
Total Units (lu + mu):          587738.17
Total Signal Nets:              28745
Total NC Nets:                  2
Average Pins/Nets:              3.616
```

An additional 112 slots on the die are used for power/ground pairs.

Section 4. Software Considerations

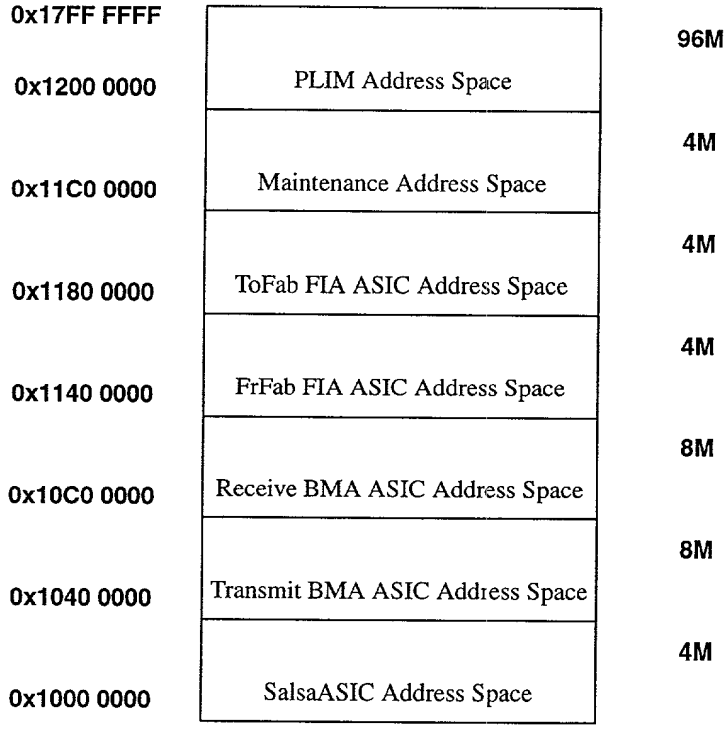
The following tables show the Address Maps for the linecard, linecard-I/O and Salsa4. There have not been any changes to these maps in this rev of Salsa.

4.1 Line-Card Address Map

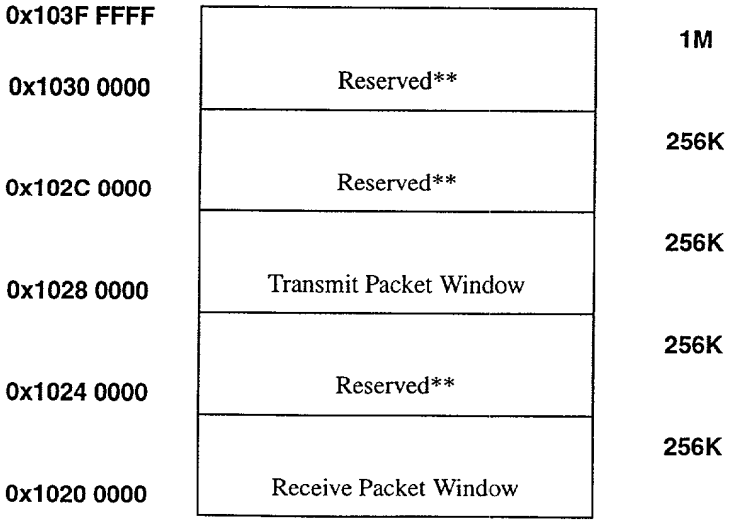


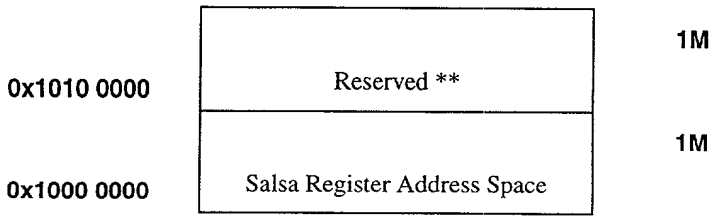
* Mapped at mupltiple locations

4.2 Line-Card I/O Address Map



4.3 Salsa ASIC Address Map





** These areas are reserved for future expansion. For this implementation version of the Salsa ASIC, accesses to these address spaces are illegal and therefore result as bus-error for read operations or erroneous interrupt for write operations

Section 5. Updated Register List

For the sake of completeness, both existing and new registers are shown below. Existing registers are shaded.

Table 4: Summary of Registers

Register Name	Access Type	Address	Section
Watch-Dog Timer	16-bit R/W	0x1000 0004	1
General-Purpose Counter	16-bit R/W	0x1000 000C	2
Real-Time Interrupt Timer	16-bit R/W	0x1000 0014	3
Receive Network Disable Timer	16-bit R/W	0x1000 001C	4
Transmit Network Disable Timer	16-bit R/W	0x1000 0024	5
Receive BMA Bus Time-Out Counter	16-bit R/W	0x1000 002C	6
Transmit BMA Bus Time-Out Counter	16-bit R/W	0x1000 0034	7
Line-card I/O Bus Time-Out Counter	16-bit R/W	0x1000 003C	8
Timers/Counters Control Register	8-bit R/W	0x1000 0044	9
Interrupt Cause Register	16-bit R	0x1000 004C	10
Interrupt Mask Register	16-bit R/W	0x1000 0054	11
Real-Time Interrupt Clear Register	8-bit W	0x1000 005C	12
Reset to BMA Register	8-bit R/W	0x1000 0064	13
Receive Packet Done Register	8-bit W	0x1000 006C	14
Transmit Packet Done Register	8-bit W	0x1000 0074	15
L3 Asic ID Register	16-bit R	0x1000 007C	16
Memory Configuration Register	16-bit R/W	0x1000 0084	17
Memory Combination Register	8-bit R/W	0x1000 008C	18
L3 Performance Enhancment Register	8-bit R/W	0x1000 0094,	19
Error Checking Enable Register	8-bit R/W	0x1000 009C,	20
Receive BMA Bus Error Status Register	8-bit R	0x1000 00A4,	21
Transmit BMA Bus Error Status Register	8-bit R	0x1000 00AC,	22
DRAM Error Status Register	8-bit R	0x1000 00B4	23
Line-card I/O Bus Error Status Register	8-bit R	0x1000 00BC	24
DRAM Catastrophic Error Address Register	32-bit R	0x1000 00C4	25

Table 4: Summary of Registers

Register Name	Access Type	Address	Section
Receive BMA Address Exception Register	32-bit R	0x1000 00CC	26
Transmit BMA Address Exception Register	32-bit R	0x1000 00D4	27
I/O Address Exception Register	32-bit R	0x1000 00DC	28
L3 Interrupt Status Information Register	8-bit R	0x1000 00E4	29
Receive BMA Packet Synopsis Register	16-bit R/W	0x1000 00F4	30
Receive BMA Hardware Assist Register	8-Bit R/W	0x1000 00FC	31
Receive BMA Buffer Service Information Register	8-bit R	0x1000 0104	32
Receive BMA Buffer Flush Information Register	8-bit R	0x1000 010C	33
Transmit BMA Buffer Service Information Register	8-bit R	0x1000 0114	34
Transmit BMA Buffer Flush Information Register	8-bit R	0x1000 011C	35
Receive BMA Packet Updated Info Register	32-bit R	0x1000 0124	36
Receive BMA Protocol 0 Identifier Register	32-bit R/W	0x1000 012C	37
Receive BMA Protocol 1 Identifier Register	32-bit R/W	0x1000 0134	38
Receive BMA Protocol 2 Identifier Register	32-bit R/W	0x1000 013C	39
Receive BMA Protocol 3 Identifier Register	32-bit R/W	0x1000 0144	40
FIB Root Regsiter	32-bit R/W	0x1000 014C	41
Leaf Pointer Register	32-bit R	0x1000 0154	42
Lookup Result Register	32-bit R	0x1000 015C	43
Diagnostic access to Receive BMA Prefetch Buffer 0	64-bit R/W (byte WR)	1000_1000 - 1000_105F	
Diagnostic access to Receive BMA Prefetch Buffer 1	64-bit R/W (byte WR)	1000_1080 - 1000_10DF	
Diagnostic access to Receive BMA Prefetch Buffer 2	64-bit R/W (byte WR)	1000_1100 - 1000_115F	
Diagnostic access to Transmit BMA Prefetch Buffer 0	64-bit R/W (byte WR)	1000_1200 - 1000_125F	

Table 4: Summary of Registers

Register Name	Access Type	Address	Section
Diagnostic access to Transmit BMA Prefetch Buffer 1	64-bit R/W (byte WR)	1000_1280 - 1000_12DF	
Diagnostic access to Transmit BMA Prefetch Buffer 2	64-bit R/W (byte WR)	1000_1300 - 1000_135F	
Diagnostic access to DRAM Write Buffer	64-bit R/W	1000_1500 - 1000_15BF	
Diagnostic access to Receive BMA Write Buffer	64-bit R/W	1000_1600 - 1000_16BF	
Diagnostic access to Transmit BMA Write Buffer	64-bit R/W	1000_1700 - 1000_17BF	
Port0 ACL Tree Base Register	32bit R/W	0x1000 0164	44
Port1 ACL Tree Base Register	32bit R/W	0x1000 016C	45
Port2 ACL Tree Base Register	32bit R/W	0x1000 0174	46
Port3 ACL Tree Base Register	32bit R/W	0x1000 017C	47
Port4 ACL Tree Base Register	32bit R/W	0x1000 0184	48
Port5 ACL Tree Base Register	32bit R/W	0x1000 018C	49
Port6 ACL Tree Base Register	32bit R/W	0x1000 0194	50
Port7 ACL Tree Base Register	32bit R/W	0x1000 019C	51
ACL Engine Config Register	16bit R/W	0x1000 01A4	52
Current ACL Node HI Bytes Register	32bit R	0x1000 01AC	53
Current ACL Node LO Bytes Register	32bit R	0x1000 01B4	54
Current ACL Hash Key Register	8bit R	0x1000 01BC	55
ACL Engine Status Register	32bit R	0x1000 01C4	56
ACL Hash Key Selection Register	32bit R/W	0x1000 01CC	57
ACL Hash Key Selection Register	32bit R/W	0x1000 01D4	58
ACL Hash Key Selection Register	16bit R/W	0x1000 01DC	59
ACL Compare Mask 0 Register	32bit R/W	0x1000 01E4	60
ACL Compare Mask 1 Register	32bit R/W	0x1000 01EC	61
ACL Compare Mask 2 Register	32bit R/W	0x1000 01F4	62
ACL Compare Mask 3 Register	32bit R/W	0x1000 01FC	63
ACL Compare Mask 4 Register	32bit R/W	0x1000 0204	64
ACL Compare Mask 5 Register	32bit R/W	0x1000 020C	65

Table 4: Summary of Registers

Register Name	Access Type	Address	Section
ACL Compare Mask 6 Register	32bit R/W	0x1000 0214	66
ACL Compare Mask 7 Register	32bit R/W	0x1000 021C	67
ACL Compare Mask 8 Register	32bit R/W	0x1000 0224	68
ACL Compare Mask 9 Register	32bit R/W	0x1000 022C	69
ACL Max Nodes Register	16bit R/W	0x1000 0234	70
ACL Current Count Register	16bit R	0x1000 023C	71
ECC Status Register	16bit R	0x1000 0244	72
ECC Diagnostic Syndrome Register	8bit R/W	0x1000 024C	73
ECC Config Register	8bit R/W	0x1000 0254	74
ECC SBE Address Register	32bit R	0x1000 025C	75
ECC SBE Syndrome Register	8bit R	0x1000 0264	76
ECC MBE Syndrome Register	8bit R	0x1000 026C	77

There are eight count-down counters implemented in the Salsa ASIC to serve various purposes. All of these counters are 16-bit read/write. On power-on reset, all counters are disabled and loaded with a default count of 65,535 (0xFFFF). Under software control, each counter can be loaded with a known value and enabled through proper write operations. Each timer will count as many usec clocks as the programmed value (i.e. a programmed value of 5, will produce a timed interval of 4 to 5 usecs). All reads & writes to the counters must be 16-bit accesses.

5.1 Watch-Dog Timer

16-bit R/W
0x1000 0004

After reset, the timer is disabled and loaded with a default value of 0xFFFF, i.e. 65.535 msecs.
Start values other than the default 0xFFFF, can be programmed by writing to this address. If enabled the timer will automatically decrement, every 1us if there is a non-zero value in the counter. The enable bit can be found in the "Timers/Counters Control Register" on page 24.

Should the timer reach 0x0000, an NMI interrupt will be generated and the timer stops counting. To clear the NMI flag, software must program a new value to the Watch Dog timer. A value of 0x0000 is acceptable if it is desired not to restart the timer.

At any time, the Watch Dog Timer Enable may be switched off, At that point, any value can be written to the Watch Dog timer, without triggering the decrement logic. Note: This timer will not generate a reset at

end of count, as previously proposed.

5.2 General-Purpose Counter

16-bit R/W
0x1000 000C

After reset, the counter is disabled and loaded with a default value of 0xFFFF.

The counter is enabled and disabled through the "Timers/Counters Control Register" on page 24. A predetermined count-down value can be loaded with a 16-bit write operation at this address. Otherwise, an after-reset default value of 0xFFFF will be assumed.

Should the timer reach 0x0000, an interrupt will be generated and the timer stops counting. To clear the interrupt flag, software must program a new value to the General Purpose counter. A value of 0x0000 is acceptable if it is desired not to restart the timer. At any time, the General Purpose counter Enable may be switched off, Once this is done, any value can be written to the timer, without triggering the decrement logic. *Note: This is different from the IRSP interrupt scheme. Software does not need to keep polling this counter!*

5.3 Real-Time Interrupt Timer

16-bit R/W
0x1000 0014

After reset, the timer is disabled and loaded with a default value of 0xFFFF.

The timer is enabled and disabled through the "Timers/Counters Control Register" on page 24. A predetermined count can be loaded with a 16-bit write operation at this address. Otherwise, an after-reset default value of 0xFFFF will be assumed.

If the counter is enabled, a decrement occurs once every usec. It will count down until it is disabled or when it reaches a value of 0x0001. As a result, an interrupt will be latched and issued. On the next clock the timer will be reloaded with the previous loaded value. Software needs to acknowledge the interrupt and clear the interrupt latch by performing a write operation to the "Real-Time Interrupt Clear Register" on page 26.

5.4 Receive Network Disable Timer

16-bit R/W
0x1000 001C

After reset, the timer is disabled and loaded with a default value of 0x0000.

This special timer is used to disable the *receive network interrupt* for a period of time. Software must write a 16-bit value to this register, causing it to start decrementing once every 20ns if the timer is enabled. During the decrement, the receive network interrupt will be blocked. When the timer value reaches zero, it will remain there and un-block the receive network interrupt. The timer will not be activated until software

writes to it again. Note: This timer does not generate an interrupt!

5.5 Transmit Network Disable Timer

16-bit R/W
0x1000 0024

After reset, the timer is disabled and loaded with a default value of 0x0000.

This special timer is used to disable the transmit network interrupt for a period of time. Software must write a 16-bit value to this register, causing it to start decrementing once every 20ns if the timer is enabled. During the decrement, the transmit network interrupt will be blocked. When the timer value reaches zero, it will remain there and un-block the transmit network interrupt. The timer will not be activated until software writes to it again. Note: This timer does not generate an interrupt!

5.6 Receive BMA Bus Time-Out Counter

16-bit R/W
0x1000 002C

After reset, the timer is disabled and loaded with a default value of 0xFFFF.

A predetermined count can be loaded with a 16-bit write operation at the timer's address. The timer, however, will not be activated until a legal operation is initiated on the Receive BMA Bus by the L3 ASIC. The timer will be reloaded with the programmed count value and start counting down at a rate of one microseconds immediately after the Request cycle. It will stop counting if the counter is exhausted or when the bus transaction ends, i.e. when the **bma_req_l** goes away. If the timer expires before the end of the bus transaction, a corresponding error bit will be set in the Receive BMA Bus Error Status Register. Depending on the operation type, a bus error or an erroneous interrupt is posted to the P4 processor. *Note: The count value should be selected to be larger than the longest bus transaction, i.e. the prefetch operation.*

5.7 Transmit BMA Bus Time-Out Counter

16-bit R/W
0x1000 0034

After reset, the timer is disabled and loaded with a default value of 0xFFFF.

A predetermined count can be loaded with a 16-bit write operation at the timer's address. The timer, however, will not be activated until a legal operation is initiated on the Transmit BMA Bus by the L3 ASIC. The timer will be reloaded with the programmed count value and start counting down at a rate of one microseconds immediately after the Request cycle. It will stop counting if the counter is exhausted or when the bus transaction ends, i.e. when the **bma_req_l** goes away. If the timer expires before the end of the bus transaction, a corresponding error bit will be set in the Transmit BMA Bus Error Status Register. Depending on the operation type, a bus error or an erroneous interrupt is posted to the P4 processor. *Note: The count value should be selected to be larger than the longest bus transaction, i.e. the prefetch operation.*

5.8 Line-card I/O Bus Time-Out Counter

16-bit R/W
0x1000 003C

After reset, the timer is disabled and loaded with a default value of 0xFFFF.

The timer is enabled or disabled through its enable bit in the Counter Control Register @0x1000 0044. A predetermined count can be loaded with a 16-bit write operation at the timer's address. The timer, however, will not be activated until a legal operation is initiated on the Line-Card I/O Bus by the L3 ASIC. The timer will be reloaded with the programmed count value and start counting down at a rate of one microseconds immediately after the Request cycle. It will stop counting if the counter is exhausted or when the bus transaction ends, i.e. when the **io_strobe_l** goes away. If the timer expires before the end of the bus transaction, a corresponding error bit will be set in the Line-Card I/O Bus Error Status Register. Depending on the operation type, a bus error or an erroneous interrupt is posted to the P4 processor. Note: The count value should be selected to be larger than the longest bus transaction.

5.9 Timers/Counters Control Register

8-bit R/W
0x1000 0044

After reset, the register is loaded with a default value of 0x00.

The register is read/write through 8-bit operations. Immediately after reset, the P4 processor needs to write to this register with bit 2 set to "1" in order to enable the real-time interrupt.

Bit	Description
4	Watch Dog Timer Enable 0 - Disabled 1- Enable
3	General-Purpose Counter Enable 0 - Disabled 1 - Enabled
2	Real-Time Timer Enable 0 - Disabled 1 - Enabled
1	"Receive Network Disable" Timer Enable 0 - Disabled 1 - Enabled
0	"Transmit Network Disable" Timer Enable 0 - Disabled 1 - Enabled

5.10 Interrupt Cause Register

16bit R
0x1000 004C

After reset, the register is loaded with a default value of 0x0000.

This register captures the source of the general interrupt, sent to the P4.

Bit		Description
[15:14]		Reserved.
13	Indirect 8	Receive PSA ASIC (external)
12	Direct #0	Receive Network Interrupt.
11	Direct #1	Transmit Network Interrupt.
10	Direct #2	Maintenance Bus Interrupt.
9	Direct #3	L3 Interrupt.
8	Indirect #0	Receive BMA ASIC (external).
7	Indirect #1	Transmit BMA ASIC (external).
6	Indirect #2	ToFab FIA ASIC (external).
5	Indirect #3	FrFab FIA ASIC (external)
4	Indirect #4.	PLIM Interrupt Int-0 (external)
3	Indirect #5.	PLIM Interrupt Int-1 (external)
2	Indirect #6a	General Purpose Counter Interrupt.
1	Indirect #6b	Real-Timer Interrupt.
0	Indirect #7	Error Interrupt.

5.11 Interrupt Mask Register

16bit R/W
0x1000 0054

After reset, the register is loaded with a default value of 0xFFFF, i.e. masking all direct and indirect interrupt sources.

The register is read/write through a 16-bit operation. Software can individually enable any interrupts by writing a "0" into the corresponding bits of the register.

Bit	Description
[15:13]	Reserved.
12	Reveive PSA ASIC Interrupt Mask Bit
11	Receive Network Interrupt Mask Bit
10	Transmit Network Interrupt Mask Bit
9	Maintenance Bus Interrupt Mask Bit

Bit	Description
8	L3 Interrupt Mask Bit
7	Receive BMA ASIC Interrupt Mask Bit
6	Transmit BMA ASIC Interrupt Mask Bit
5	ToFab FIA ASIC Interrupt Mask Bit
4	FrFab FIA ASIC Interrupt Mask Bit
3	PLIM Interrupt-0 Mast Bit
2	PLIM Interrupt-1 Mask Bit
1	Timer Interrupt Mask Bit (masks both General Purpose and Real Time, timer interrupts
0	Error Interrupt Mask Bit

5.12 Real-Time Interrupt Clear Register

8 bit W
0x1000 005C

An 8-bit write operation to this register will clear the real-time interrupt. Data pattern is don't care.

5.13 Reset to BMA Register

8 bit R/W
0x1000 0064

After reset, the register is loaded with a default value of 0x0000.

This register is used to send a reset to the BMA logic. A level reset signal is generated from the value stored in each bit. This means that each reset must be turned on and off, by setting and clearing the corresponding bit.

Bit	Description
[7:2]	Reserved.
1	Rx BMA reset 0 = No reset 1 = Initiates a reset to the Rx BMA asic
0	Tx BMA reset 0 = No reset 1 = Initiates a reset to the Tx BMA asic

5.14 Receive Packet Done Register

8-bit W
0x1000 006C

A write operation performed on this register will clear the Receive Network Service Request bit of the in-service prefetch buffer. This in turn will set the corresponding Flush Request bit for this selected prefetch buffer. In order to preserve the P4's order, the flush request is queued into the Receive BMA write buffer. This ensures that all the previous write requests from the P4 processor will be executed before this flush request.

5.15 Transmit Packet Done Register

8 bit W
0x1000 0074

A write operation performed on this register will clear the Transmit Network Service Request bit of the in-service prefetch buffer. This in turn will set the corresponding Flush Request bit for this prefetch buffer. In order to preserve the 7P4's order, the flush request is queued into the Transmit BMA write buffer. This ensures that all the previous write requests from the P4 processor will be executed before this flush request.

5.16 L3 Asic ID Register

16 bit R
0x1000 007C

This is a read-only register. Sixteen bits of information are extracted from the JTAG ID register which has the following format. The register is always read as 16'h10C9.

Bit	Description
15:12	ASIC Revision.
11:0	ASIC Part Number.

5.17 Memory Configuration Register

16-bit R/W
0x1000 0084

After reset, the register is loaded with a default value of 026E, i.e. 2 clock CAS precharge, 5 clock RAS precharge, 4 clock WRITE tRCD, 5 clock READ tRCD, refresh rate = 15 uS.

The values on power-up are suited for a 60ns EDO. For a 50ns EDO, this register should be programmed with a value of 000E, i.e. 2 clock CAS precharge, 4 clock RAS precharge, 4 clock read and write tRCD, refresh rate = 15uS. The feature to increase tCP to 3 clocks is intended for lab use, and will cause performance degradation. In order to use this feature, tRCD must be at least 4 clocks.

Bit	Description
15:11	Reserved.
10	Clocks spent in CAS precharge, tCP, following an access. Applies to single and burst accesses. 0: 2 clock 1: 3 clocks
9:8	Clocks spent in RAS precharge (i.e RAS HI) 00: 3 clocks 01: 4 clocks 10: 5 clocks 11: 6 clocks
7:6	Clocks spent during tRCD (i.e RAS-to-CAS) on write accesses 00: 3 clocks 01: 4 clocks 10: 5 clocks 11: 6 clocks
5:4	Clocks spent during tRCD (i.e RAS-to-CAS) on read accesses 00: 3 clocks 01: 4 clocks 10: 5 clocks 11: 6 clocks
3:0	DRAM Refresh Timer Value (in microseconds) 1111 - Refresh every 16 microseconds 1110 - Refresh every 15 microseconds : 0000 - Refresh every microsecond

5.18 Memory Combination Register

8-bit R/W
0x1000 008C

After reset, the register is loaded with a default value of 00x11, i.e. Single DIMM with 16Mbytes

Bit	Description
7:6	Reserved
5	Number of DIMMs being used 0 - Single DIMM. 1 - Two DIMM..
4:0	DIMM combination must be one of

[4:0]	Dimm 0	Dimm 1
00000	8	8 or noth-
00001	16	8
00010	32	8
00011	64	8
00100	128	8
00101	256	8
N/A	512	8
00110	16	16 or
00111	32	16
01000	64	16
01001	128	16
01010	256	16
01011	512	16
01100	32	32 or
01101	64	32
01110	128	32
01111	256	32
10000	512	32
10001	64	64 or
10010	128	64
10011	256	64
10100	512	64
10101	128	128 or
10110	256	128
10111	512	128
11000	256	256 or

[4:0]	Dimm 0	Dimm 1
11001	512	256
11010	512	512

5.19 L3 Performance Enhancment Register

8-bit R/W
0x1000 0094,

After reset, this register is loaded with a default value of 0x0000, i.e. all read-around-writes and prefetch buffers are disabled.

Bit	Description
7	Reserved.
6	Receive BMA Read-Around-Write Enable 0 - Disabled 1 - Enabled
5	Transmit BMA Read-Around-Write Enable 0 - Disabled 1 - Enabled
4	Main Memory Read-Around-Write Enable 0 - Disabled 1 - Enabled
3	Enable early clear of receive net work interrupt 0 - Disabled 1 - Enabled
2	Enable early clear of transmit network interrupt 0 - Disabled 1 - Enabled
[1:0]	Reserved

5.20 Error Checking Enable Register

8-bit R/W
0x1000 009C,

After reset, this register is loaded with a default value of 0x00, i.e. all checking is disabled.

Bit	Description
[7:5]	Reserved.
4	P4 Timeout Error Checking 0 - Disable P4 timeout checking on reads taking place on the P4 bus 1 - Enable P4 timeout checking on reads taking place on the P4 bus
3	Main Memory Parity Checking & ECC Bus Error Enable 0 - Disable write data parity check by P4 for main memory data. Also Disables Bus Error during reads that show a multibit ECC error 1 - Enable read/write parity check by P4 for main memory data. Also enables Bus Errors on reads that have multibit ECC errors in them.
2	Receive BMA Error Checking 0 - Disable error check on the receivte BMA bus interface 1 - Enable error check on the receive BMA bus interface
1	Transmit BMA Error Checking 0 - Disable error check on the transmit BMA bus interface 1 - Enable error check on the transmit BMA bus interface
0	I/O Checking Enable 0 - Disable illegal Address/Data width checking on I/O interface 1 - Enable illegal Address/Data width checking on I/O interface

5.21 Receive BMA Bus Error Status Register

8-bit R
0x1000 00A4,

After reset, this register is loaded with a default value of 0x00, i.e. no error has been detected.

When a Receive BMA bus error occurs during an access, a corresponding bit will be set in this register. If the error had been due to a write operation (as indicated by bit[6] being set), an interrupt to the P4 processor will be generated. If the error had been due to a read operation, a bus-error will be returned to P4. A read from this register will clear all the bits. For all bits, a "1" value indicates an Error of the corresponding type has occurred.

Bit	Description

Bit	Description
10	Receive BMA Interrupt errors: Logical OR of bits [7], [6], [5], [2], [1] and [0].
9	Receive BMA Non-interrupt errors: Logical OR of bits [8], [4] and [3].
8	Timeout error on BMA bus, during Read
7	Timeout error on BMA bus, during Packet prefetch
6	Timeout error on BMA bus, durng Write dispatch
5	Timeout error on BMA bus, during Packet flush.
4	Bus Read Parity Error
3	Packet service (read from prefetched packet) parity error
2	Packet prefetch from Receive BMA, parity error
1	Write dispatich parity error
0	Packet Flush parity error

5.22 Transmit BMA Bus Error Status Register

0x1000 00AC,
8-bit R

After reset, this register is loaded with a default value of 0x00, i.e. no error has been detected.

When a Transmit BMA bus error occurs during an access, a corresponding bit will be set in this register. If the error had been due to a write operation (as indicated by bit[6] being set), an interrupt to the P4 processor will be generated. If the error had been due to a read operation, a bus-error will be returned to P4. A read from this register will clear all the bits. For all bits, a "1" value indicates an Error of the corresponding type has occurred.

Bit	Description
10	Transmit BMA Interrupt errors: Logical OR of bits [7], [6], [5], [2], [1] and [0].
9	Transmit BMA Non-interrupt errors: Logical OR of bits [8], [4] and [3].
8	Timeout error on BMA bus, during Read
7	Timeout error on BMA bus, during Packet prefetch
6	Timeout error on BMA bus, durng Write dispatch
5	Timeout error on BMA bus, during Packet flush.
4	Bus Read Parity Error
3	Packet service (read from prefetched packet) parity error

Bit	Description
2	Packet prefetch from Receive BMA, parity error
1	Write dispatich parity error
0	Packet Flush parity error

5.23 DRAM Error Status Register

0x1000 00B4
8-bit R

After reset, this register is loaded with a default value of 0x00, i.e. no error has been detected.

When a DRAM error occurs during an access, a corresponding bit will be set in this register. If the error had been due to a write operation (as indicated by bit[4] being set), an interrupt to the P4 processor will be generated. If the error had been due to a read operation, a bus-error will be returned to P4. A read from this register will clear all the bits.

A special note about partial writes: P4 partial writes are handled as read-modify-write operations in Salsa4. Should an uncorrectable ECC error (MBE) occur during this operation, the write is aborted and both write and read error bits are set. This register's value becomes 33 during that time.

Bit	Description
[7:6]	Reserved
5	DRAM Read Error (OR condition of bits [0] and [3], shown below) 0 - No error 1 - Error
4	DRAM Write Error (OR condition of bits [1-2], shown below) 0 - No error 1 - Error
3.	DRAM Read Address Error 0 - No Error 1 - Read address is out-of-bounds for current memory combination specified in Memory Combination Register, page 29
2	DRAM Write Address Error 0 - No Error 1 - Write address is out-of-bounds for current memory combination specified in Memory Combination Register, page 29
1	DRAM Write Parity Error 0 - No Error

Bit	Description
	1 - Error
0	DRAM Read ECC Single/Multibit Error.
	0 - No error
	1 - Either a multibit or single bit error has occurred

5.24 Line-card I/O Bus Error Status Register

0x1000 00BC
8-bit R

After reset, this register is loaded with a default value of 0x00, i.e. no error has been detected. When an I/O bus error occurs during an access, a corresponding bit will be set in this register. If the error had been due to a write operation (as indicated by bit[4] being set), an interrupt to the P4 processor will be generated. If the error had been due to a read operation, a bus-error will be returned to P4. A read from this register will clear all the bits.

Bit	Description
[7:6]	Reserved.
5	I/O Bus Read Error (OR Condition of bits [1] and [3], shown below)
	0 - No error
	1 - Error
4	I/O Bus Write Error (OR Condition of bits [0] and [2], shown below)
	0 - No error
	1 - Error
3	Line-card I/O Bus Read Time-Out Error.
	0 - No error
	1 - A bus timeout occurred before io_rdy_l was asserted
2	Line-card I/O Bus Write Time-Out Error.
	0 - No error
	1 - A bus timeout occurred before io_rdy_l was asserted
1	Illegal Line-card I/O Read Accessing Error
	0 - No error
	1 - An invalid read operation is attempted by P4. Please see the summary of illegal I/O bus operatons listed in Chapter 3 of the BFR Quad OC3 Linecard spec. (ENG-7439).

Bit	Description
0	Illegal Line-card I/O Write Accessing Error 0 - No error 1 - An invalid write operation is attempted by P4. Please see the summary of

5.25 DRAM Catastrophic Error Address Register

0x1000 00C4
32-bit R

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

This register captures the 32bit address at the first occurrence of any error (since its last reading) of the "DRAM Error Status Register" on page 33, except for SBE's. A separate address register for single-bit-errors can be found in "ECC SBE Address Register" on page 52.

Bit	Description
[31:0]	Address Bits [31:0]

5.26 Receive BMA Address Exception Register

0x1000 00CC
32-bit R

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

This register will capture 32 bits of the address at which the error occured

Bit	Description
[31:0]	Address Bits [31:0]

5.27 Transmit BMA Address Exception Register

0x1000 00D4
32-bit R

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

This register will capture 32 bits of the address at which the error occured.

Bit	Description
[31:0]	Address Bits [31:0]

5.28 I/O Address Exception Register

0x1000 00DC
32-bit R

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.
This register will capture the 32 bits of the address at which the error occured.

Bit	Description
[31:0]	Address Bits [31:0]

5.29 L3 Interrupt Status Information Register

0x1000 00E4
8-bit R

This register further breaks down the cause of the L3 Error Interrupt bit, bit 0[, in "Interrupt Cause Register" on page 25. All causes listed here are from internal L3 Error checking and timer logic.

Bit	Description
[7:5]	Reserved
4	rx_bma_error_int
3	tx_bma_error_int
2	dram_error_int
1	io_error_int
0	General Purpose or Real-Time timers have expired

5.30 Receive BMA Packet Synopsis Register

16bit R/W
0x1000 00F4

After reset, the value of this register is all zeros.

Only bit[4] of this register is write accessible. Any upper bits are ignored during a P4 write operation.

Bit	Description
15:11	Reserved

Bit	Description
10	Copy of THB bit from BHDR
9	0 - Packet passed length check 1 - Error: Packet failed length check
8	0 - Packet is IPv4, no options 1 - Error: This is not a "fast-path" packet
7	0 - Protocol identifier in MAC matches matches expected value 1 - Error: Unknown protocol identifier in MAC
6	0 - Prior to decrement TTL value was larger than 1 1 - Error: Packet TTL is 1 or 0
5	0 - Checksum validation passed 1 - Error: Incorrect checksum detected
4	0 - Disable hardware generated TTL and Checksum updates when flushing the current packet 1 - Okay to use hardware generated TTL and Checksum when flushing the current packet Note, this disable automatically clears with each new packet.
3	A sampling of the MTRIE lookup enable bit in register "Receive BMA Hardware Assist Register" on page 37, at the time the lookup starts.
2	0 - No out-of-range error during lookup reads 1 - Error: Encountered an out-of-range read address during lookup read
1	0 - No parity error 1 - Error: A parity mismatch occurred during MTRIE lookup
0	0 - MTRIE lookup completed successfully 1 - Error: Could not complete MTRIE lookup because no leaf was found after using all octets of IP destination address

5.31 Receive BMA Hardware Assist Register

8-Bit R/W
0x1000 00FC

After reset, this register contains the value 04.

This register enables/disables MTRIE lookup and/or hardware TTL decrement and Checksum updates for all packets. Please note that bit[4] of the "Receive BMA Packet Synopsis Register" disables only on a per-packet basis whereas bit[0] here, disables TTL and checksum updates for all packets.

Bit	Description
7:3	Reserved

Bit	Description
2	0 - Disable 1 - Enable 64 byte reversed prefetch from BMA. The value of Bit[1] is ignored if this bit is 0, and MTRIE lookup is disabled.
1	0 - Disable 1 - Enable MTRIE lookup by hardware
0	0 - Disable 1 - Enable hardware TTL decrement and Checksum recalculation

5.32 Receive BMA Buffer Service Information Register

8-bit R
0x1000 0104

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

For this buffer there are two associated flags: the Valid Service Request bit and the Flush Request bit. The Valid Service Request bit is set when a buffer is completely filled with the prefetched packet information from the corresponding BMA logic and ready to be processed by the P4 processor. The Flush Request bit of a buffer is set when the microcode is done with the currently selected buffer and wants to have its of data flushed back to the corresponding BMA logic. A buffer is available only when its associated Valid Service Request and Flush Request flags are not set, i.e. "0".

Bit	Description
[7]	Reserved.
[6:5]	Prefetch Buffer Selector 00 = Buffer#0 is currently being selected for prefetching. 01 = Buffer#1 is currently being selected for prefetching. 10 = Buffer#2 is currently being selected for prefetching. 11 = No buffer is selected for prefetching. (equivalent to having the prefetching buffers disabled.
4	Valid Service Request flag for Buffer#2 0 - No service needed 1 - Has packet information data, needs service
3	Valid Service Request flag for Buffer#1 0 - No service needed 1 - Has packet information data, needs service

Bit	Description
2	Valid Service Request flag for Buffer#0 0 - No service needed 1 - Has packet information data, needs service
[1:0]	In-service Buffer Selector 00 = Buffer#0 is currently being selected for in-service process. 01 = Buffer#1 is currently being selected for in-service process. 10 = Buffer#2 is currently being selected for in-service process. 11 = No buffer is selected for in-service. Note: When the selector is set to "11", i.e. no buffer is selected for in-service, it is equivalent to having the in-service buffers disabled

The Received Network Interrupt is asserted when any of the Valid Service Request flags are set. The Valid Service Request flag for the selected in-service buffer will be cleared when the P4 writes to the Packet Done register.

5.33 Receive BMA Buffer Flush Information Register

8-bit R
0x1000 010C

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

Bit	Description
[7:5]	Reserved.
[4:3]	Flush Buffer Selector 00 = Buffer#0 is currently being selected for flushing. 01 = Buffer#1 is currently being selected for flushing. 10 = Buffer#2 is currently being selected for flushing. 11 = No buffer is selected for flushing.
2	Flush Request flag for Flush Buffer #2. 0 = Inactive 1 = Has data, needs to be flushed back to the received BMA logic.
1	Flush Request Bit for Prefetch Buffer #1. 0 = Inactive

Bit	Description
	1 = Has data, needs to be flushed back to the received BMA logic.
0	Flush Service Request Bit for Prefetch Buffer #0. 0 = Inactive

5.34 Transmit BMA Buffer Service Information Register

8-bit R
0x1000 0114

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

Bit	Description
[7]	Reserved.
[6:5]	Prefetch Buffer Selector 00 = Buffer#0 is currently being selected for prefetching. 01 = Buffer#1 is currently being selected for prefetching. 10 = Buffer#2 is currently being selected for prefetching. 11 = No buffer is selected for prefetching., equivalent to having the prefetching buffers disabled.
4	Valid Service Request flag for Buffer#2 0 - No service needed 1 - Has packet information data, needs service
3	Valid Service Request flag for Buffer#1 0 - No service needed 1 - Has packet information data, needs service
2	Valid Service Request flag for Buffer#0 0 - No service needed 1 - Has packet information data, needs service
[1:0]	In-service Buffer Selector 00 = Buffer#0 is currently being selected for in-service process. 01 = Buffer#1 is currently being selected for in-service process. 10 = Buffer#2 is currently being selected for in-service process.

Bit	Description
	11 = No buffer is selected for in-service. When the selector is set to "11", i.e. no buffer is selected for in-service, it is equivalent to having the in-service buffers disabled.

The Transmitted Network Interrupt is asserted when any of the Valid Service Request flags are set. The Valid Service Request flag for the selected in-service buffer will be cleared when the P4 writes to the Packet Done register.

5.35 Transmit BMA Buffer Flush Information Register

8-bit R
0x1000 011C

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

Bit	Description
[7:5]	Reserved.
[4:3]	Flush Buffer Selector
	00 = Buffer#0 is currently being selected for flushing.
	01 = Buffer#1 is currently being selected for flushing.
	10 = Buffer#2 is currently being selected for flushing.
	11 = No buffer is selected for flushing.
2	Flush Request flag for Flush Buffer #2.
	0 = Inactive
	1 = Has data, needs to be flushed back to the transmit BMA logic.
1	Flush Request Bit for Prefetch Buffer #1.
	0 = Inactive
	1 = Has data, needs to be flushed back to the transmit BMA logic.
0	Flush Service Request Bit for Prefetch Buffer #0.
	0 = Inactive
	1 = Has data, needs to be flushed back to the transmit BMA logic.

5.36 Receive BMA Packet Updated Info Register

32 bit R
0x1000 0124

After reset the contents of this register are indeterminate. After an Rx packet service request is issued, this register will hold values relevant to the current packet being pointed to in the "Receive BMA Buffer Service Information Register" on page 38.

Bit	Description
31:24	Updated Time-to-live
23:16	Reserved
15:0	Updated header checksum

5.37 Receive BMA Protocol 0 Identifier Register

32bit R/W
0x1000 012C

After reset, the register has an indeterminate value.

This register contains the value of a protocol identifier that is compared against bits [31:0] of the MAC header field in the Receive Packet Window.

Bit	Description
31:0	The value to check for, in the prefetched packet's MAC header.

5.38 Receive BMA Protocol 1 Identifier Register

32bit R/W
0x1000 0134

After reset, the register has an indeterminate value.

This register contains the value of a protocol identifier that is compared against bits [31:0] of the MAC header field in the Receive Packet Window.

Bit	Description
31:0	The value to check for, in the prefetched packet's MAC header.

5.39 Receive BMA Protocol 2 Identifier Register

32bit R/W
0x1000 013C

After reset, the register has an indeterminate value.

This register contains the value of a protocol identifier that is compared against bits [31:0] of the MAC header field in the Receive Packet Window.

Bit	Description
31:0	The value to check for, in the prefetched packet's MAC header.

5.40 Receive BMA Protocol 3 Identifier Register

32bit R/W
0x1000 0144

After reset, the register has an indeterminate value.

This register contains the value of a protocol identifier that is compared against bits [31:0] of the MAC header field in the Receive Packet Window.

Bit	Description
31:0	The value to check for, in the prefetched packet's MAC header.

5.41 FIB Root Register

32bit R/W
0x1000 014C

After reset the register has an indeterminate value.

This register is the Base Address for all MTRIE lookups.

Bit	Description
31:24	Reserved.
23:0	FIB Root address

5.42 Leaf Pointer Register

32bit R
0x1000 0154

After reset the value of this register is all zeros.

This register holds one of two values. For a successful MTRIE lookup with all checks passed, the value in this register is the leaf pointer for the current packet in service. The checks are:

1. THB = 0
2. Packet Length check passed
3. Packet is IPv4 with no-options
4. Identifiable protocol field in encapsulation MAC header
5. TTL is larger than 1
6. Checksum validation
7. MTRIE lookup was enabled at the time of lookup
8. No parity errors during lookup
9. A leaf was found within 3 lookup attempts.

If any of the checks fail, the value in this register is all zeros.

Bit	Description
31:1	bits [31:1] of the leaf pointer read from MTRIE lookup or all zeros
0	Always zero

5.43 Lookup Results Register

32bit R
0x1000 015C

After reset the value of this register is all zeros.

This is the last read value from MTRIE lookup, before the packet was given for service to the P4. If no errors occurred during hardware assist, this register will contain the Leaf pointer, equivalent to the "Leaf Pointer Register", above.

Bit	Description
31:0	bits [31:1] of the last read from MTRIE lookup

5.44 Port0 ACL Tree Base Register

32bit R/W
0x1000 0164

5.45 Port1 ACL Tree Base Register

32bit R/W
0x1000 016C

5.46 Port2 ACL Tree Base Register

32bit R/W
0x1000 0174

5.47 Port3 ACL Tree Base Register

32bit R/W
0x1000 017C

5.48 Port4 ACL Tree Base Register

32bit R/W
0x1000 0184

5.49 Port5 ACL Tree Base Register

32bit R/W
0x1000 018C

5.50 Port6 ACL Tree Base Register

32bit R/W
0x1000 0194

5.51 Port7 ACL Tree Base Register

32bit R/W
0x1000 019C

After reset the value of the above 8 registers will be zeros

Bit	Description
31:21	Contains the 13 bit based address of the ACL tree of that port
19:0	Reserved -- read as zeros

5.52 ACL Engine Config Register

16bit R/W
0x1000 01A4

After reset the value of this register is all zeros

Bit	Description
15:2	Reserved -- read as zeros
1	Port-info location in packet 0 - port information is located in bits [25:24] of word[5] of the BHDR 1 - port information is located in bits [8:6] of the 4byte POS-like MAC header
0	ACL Engine enable 0 - ACL engine is disabled 1 - ACL engine is enabled

5.53 Current ACL Node HI Bytes Register

32bit R
0x1000 01AC

After reset the value of this register is all zeros.

Bit	Description
31:0	Bits [63:32] of the most recently read ACL node

5.54 Current ACL Node LO Bytes Register

32bit R
0x1000 01B4

After reset the value of this register is all zeros.

Bit	Description
31:0	Bits [31:0] of the most recently read ACL node

5.55 Current ACL Hash Key Register

16bit R
0x1000 01BC

After reset the value of this register is all zeros.

Bit	Description
15:10	Reserved; Read as zeros
9:0	The value of the key used in the most recent ACL Hash table lookup

5.56 ACL Engine Status Register

32bit R
0x1000 01C4

After reset the value of this register is all zeros.

Bit	Description
31:4	Bits copied from the operand field of the STOP node
3	Permit/Deny bit. This bit is also copied off bit[3] of the operand field of the STOP node. 0 - Permit 1 - Deny
2	ACL Engine out-of-range address error 0 - No error 1 - An out-of-range DRAM address error has been encountered. See the "DRAM Catastrophic Error Address Register" for the offending address
1	ACL Engine uncorrected ECC error 0 - No error 1 - The ACL engine encountered a uncorrectable ECC error during a DRAM read. See the "DRAM Catastrophic Error Address Register" for address information
0	ACL Engine maxed out error 0 - The ACL engine completed successfully. 1 - The ACL engine did NOT reach a permit/deny node at the end of the maximum number of lookups

Bits [3:0] of this register are all zeros for a packet that has completed ACL lookup and is permitted.

5.57 ACL Hash Key LO Selection Register

32bit R/W
0x1000 01CC

After reset the value of this register is **32'h00000000**.

Bit	Description
31:24	Selection bits for bit[3] of the hash key.
22:16	Selection bits for bit[2] of the hash key.
15:8	Selection bits for bit[1] of the hash key.
7:0	Selection bits for bit[0] of the hash key.

These selection bits are used to construct the hashkey, one bit at a time. Each byte selects one of the 256 bits in the 32bytes immediately following the BHDR.

5.58 ACL Hash Key MID Selection Register

16bit R/W
0x1000 01D4

After reset the value of this register is **32'h0000**.

Bit	Description
31:24	Selection bits for bit[7] of the hash key.
22:16	Selection bits for bit[6] of the hash key.
15:8	Selection bits for bit[5] of the hash key.
7:0	Selection bits for bit[4] of the hash key.

These selection bits are used to construct the hashkey, one bit at a time. Each byte selects one of the 256 bits in the 32bytes immediately following the BHDR.

5.59 ACL Hash Key HI Selection Register

16bit R/W
0x1000 01DC

After reset the value of this register is **16'h0000**.

Bit	Description
15:8	Selection bits for bit[9] of the hash key.
7:0	Selection bits for bit[8] of the hash key.

These selection bits are used to construct the hashkey, one bit at a time. Each byte selects one of the 256 bits in the 32bytes immediately following the BHDR.

5.60 ACL Compare Mask 0 Register

32bit R/W
0x1000 01E4

5.61 ACL Compare Mask 1 Register

32bit R/W
0x1000 01EC

5.62 ACL Compare Mask 2 Register

32bit R/W
0x1000 01F4

5.63 ACL Compare Mask 3 Register

32bit R/W
0x1000 01FC

5.64 ACL Compare Mask 4 Register

32bit R/W
0x1000 0204

5.65 ACL Compare Mask 5 Register

32bit R/W
0x1000 020C

5.66 ACL Compare Mask 6 Register

32bit R/W
0x1000 0214

5.67 ACL Compare Mask 7 Register

32bit R/W
0x1000 021C

5.68 ACL Compare Mask 8 Register

32bit R/W
0x1000 0224

5.69 ACL Compare Mask 9 Register

32bit R/W
0x1000 022C

After reset the value of this register is all zeros.

Bit	Description
31:0	32 bit mask set used during the comparison of ACL Nodes. Each bit value of 1 masks the compare of the corresponding bit in the ACL node OPERAND field.

5.70 ACL Max Nodes Register

16bit R/W
0x1000 0234

After reset the value of this register is 0.

Bit	Description
15:10	Reserved. Read as zeros
9:0	Maximum number of ACL node lookups, before the packet is handed over to software, for completion.

As a special note for TTM projects, the following calculations suggest a value to program to this register:

```
TTM linecard TX rate.....................................650Kpps
Salsa engine max performance.............................850Kpps
Max per-packet delay that can be tolerated by Salsa
engine, before it becomes the bottleneck.................5000ns
Each ACL lookup takes 160ns (includes DRAM precharge)
Max Number of ACL lookups within this time...............32
```

5.71 ACL Current Count Register

16bit R
0x1000 023C

After reset the value of this register is 0.

Bit	Description
15:10	Reserved. Read as zeros
9:0	The actual number of ACL node lookups, before the current packet was passed on to software

5.72 ECC Status Register

16bit R
0x1000 0244

After reset the value of this register is 0.

Bit	Description
15:2	Reserved. Read as zeros
1	Single-bit error on read 0 - No error 1 - Error occurred
0	Multiple-bit error on read 0 - No error 1 - Error occurred

All error status bits in this register clear upon read.

5.73 ECC Diagnostic Syndrome Register

8bit R/W
0x1000 024C

After reset the value of this register is 0.

Bit	Description
7:0	Provides direct write access to syndrome register. The contents of this register will be written to DRAM if bit [1] of the "ECC Config Register" has been set.

5.74 ECC Config Register

8bit R/W
0x1000 0254

After reset the value of this register is 8'b0000_0101.

Bit	Description
7:3	Reserved. Read as zeros

Bit	Description
2	Single bit error correction 0 - Disabled 1 - Enabled
1	Single bit error notification 0 - Single bit read errors will be silently corrected and forwared to the processor 1 - Single bit read errors will cause an interrupt. Errors will be corrected as data is forwarded to the processor
0	Use hardware-generated syndrome 0 - The value in the "ECC Diagnostic Syndrome Register" is used during DRAM writes. This is NOT the normal mode of operation, and should only be used during diagnostic testing. 1 - A hardware-generated syndrome is used for writes to DRAM.

5.75 ECC SBE Address Register

32bit R
0x1000 025C

After reset the value of this register is all zeros.

Bit	Description
31:0	Contains the address of the first SBE since the "ECC Status Register" was last read.

5.76 ECC SBE Syndrome Register

8bit R
0x1000 0264

After reset the value of this register is all zeros.

Bit	Description
7:0	Contains the syndrome of the first SBE since the "ECC Status Register" was last read.

5.77 ECC MBE Syndrome Register

8bit R
0x1000 026C

After reset the value of this register is all zeros.

Bit	Description
7:0	Contains the syndrome of the first MBE since the "ECC Status Register" was last read.

Appendix A Perl program used to choose a hashing key.

To better understand this problem, a Perl program was written to calculate the number of checks a packet would encounter based on the hash key generated from its fields. The input to the perl program is a customer ACL. The results after this program are based on a smaller hash size of 256buckets, and was based on a 315 entry ACL from AOL.

```
#!/sw/current/hppabin/perl5

######################################################################
#                                                                    #
# acl2tree.p   A perl5 program to traverse an ACL and determine      #
#              what the worst delays would be for a hardware         #
#              tree lookup algorithm.                                #
#              acl2tree.p <ACL file>                                 #
#                                                                    #
######################################################################


# get the options handling package.
require "getopts.pl";
# define the options that we expect.
&Getopts('v');

# check that the options that we need have been set.
if ($#ARGV < 0) {
    print "
 ACL2TREE.p  A perl5 program to traverse an ACL and determine what the
             worst delays would be for a hardware tree lookup algorithm.
             acl2tree.p -v <ACL file>, where
                -v Produce verbose output
    \n";
    exit 1;
    }


use POSIX;

%hoprot = (
  udp => 17,
  tcp => 6,
  ip => 4,
  ipinip => 4,
  icmp => 1,
  igrp => 9,
  gre => 47,
  igmp => 2,
  nos => 94,
  ospf => 89,
  );

%holabels = (
```

```
    bgp => 179,
    biff => 512,
    bootpc => 68,
    bootps => 67,
    chargen => 19,
    cmd => 514,
    daytime => 13,
    discard => 9,
    dnsix => 195,
    domain => 53,
    echo => 7,
    "echo-reply" => 8,
    exec => 512,
    finger => 79,
    ftp => 21,
    "ftp-data" => 20,
    gopher => 70,
    hostname => 101,
    ident => 113,
    irc => 194,
    klogin => 543,
    kshell => 544,
    log => 513,
    login => 513,
    lpd => 515,
    mobile-ip => 434,
    nameserver => 42,
    "netbios-dgm" => 138,
    "netbios-ns" => 137,
    ntp => 118,
    nntp => 119,
    "packet-too-big" => 32,
    pop2 => 109,
    pop3 => 110,
    rip => 520,
    smtp => 25,
    snmp => 26,
    snmptrap => 27,
    sunrpc => 111,
    syslog => 514,
    tacacs => 49,
    talk => 517,
    telnet => 23,
    tftp => 221,
    time => 37,
    uucp => 540,
    whois => 43,
    www => 80,
    xdmcp => 177,
    );

##############################################################################
#  Initializations                                                           #
##############################################################################

$hwchecks = 2;
$filename = $ARGV[0];
$nbits = 88;
$bmax = 88;
```

```
$bmin = 88;
$bind = 16;
$outbits = 12;
$endhash = (2**$outbits);
for ($i=0; $i < $nbits; $i++) {
   $min[$i] = $i;
   $max[$i] = 0;
   $zerocount[$i] = 0;
   $onecount[$i] = 0;
   $outstr[$i] = '0';
}


#########################################################################
#  Count the 0 and 1 occurrences of each of the 88 bits                 #
#  by parsing file once                                                 #
#########################################################################

open (ACLFILE, $ARGV[0]) or die;

while (<ACLFILE>) {
   if ($_ !~ /^[ ]*access-list/) {
      #silently ignore line
      next;
      }
   elsif (!(/udp/ || /tcp/ || /ip/ || /icmp/ || /igrp/ || /gre/ ||
        /igmp/ || /nos/ || /ospf/)) {
      print ("ERROR: Unknown protocol type in next line!\n$_\n");
      die;
      }
   #implicit else condition is to continue..
   chop;
   @aclarr = parse_current_line($_);


   # Hari's code...
   # cycle from 0 to nbits and update zero/one counts based on (0,1,x)
   # also keep track of min/max of zero/one counts
   for ($i = 0; $i < $nbits; $i++) {
       if ($aclarr[$i] eq 'x'){
          $zerocount[$i]++;
          $onecount[$i]++;
       } elsif ($aclarr[$i] eq '0'){
          $zerocount[$i]++;
       } elsif ($aclarr[$i] eq '1'){
          $onecount[$i]++;
       } else {
          print ("hash_min_max: Unknown character aclarr[$i] = $aclarr[$i]\n");
          die;
       }
       $min[$i] = ($zerocount[$i] < $onecount[$i]) ?
                  $zerocount[$i] : $onecount[$i];
       $max[$i] = ($zerocount[$i] > $onecount[$i]) ?
                  $zerocount[$i] : $onecount[$i];
       }
   }


#########################################################################
```

```
# Determine which bits occurred as 0 and 1 most evenly. These are   #
# good candidates for the key. (Hari's code)                        #
#####################################################################

# Select best bits with min/max values
for ($i = 0; $i < $nbits; $i++) {
#
# concatenate bitstrings(16 bits) of max, min, index
# sort them and then use index to set the outstr bits correctly
#
   $fixmax = substr((unpack("B*", pack("n", $max[$i]))), -$bind, $bind);
   $fixmin = substr((unpack("B*", pack("n", $min[$i]))), -$bind, $bind);
   $fixind = substr((unpack("B*", pack("n", $i))), -$bind, $bind);
   $presort[$i] = $fixmax.$fixmin.$fixind;
   print("for max=$max[$i], min=$min[$i] and i=$i: $presort[$i]\n");
   print("   where, fixmax = $fixmax, fixmin=$fixmin\n");
}
@postsort = sort(@presort);
#
# get index from post sorted array and set those bits to 1 in outstr
#
for ($i=0; $i < $outbits; $i++) {
   $bitpos = bintodec(substr($postsort[$i], -$bind, $bind));
   $outstr[$bitpos] = '1';
}

# print min/max results
#
foreach $w (@outstr) {
   $bitmask = $bitmask.$w;
}
print ("Selected bitmask=$bitmask\n");

#
# Extra prints that can be put under verbose
#
if ($opt_v) {
   print ("\nMin=");
   for ($i=0; $i < $nbits; $i++) {
      print("$i=$min[$i] ");
      }
   print ("\nMax=");
   for ($i=0; $i < $nbits; $i++) {
      print("$i=$max[$i] ");
      }
   print ("\nOnecount=");
   for ($i=0; $i < $nbits; $i++) {
      print("$i=$onecount[$i] ");
      }
   print ("\nZerocount=");
   for ($i=0; $i < $nbits; $i++) {
      print("$i=$zerocount[$i] ");
      }
   print ("\n");
   }


#####################################################################
# Using the 8 selected bits, form a key for each ACL entry that     #
```

```
# defines where in the hash table to add that entry                    #
#########################################################################

close (ACLFILE);
open (ACLFILE, $ARGV[0]) or die;

while (<ACLFILE>) {
   if ($_ !~ /^[ ]*access-list/) {
      #silently ignore line
      next;
      }
   #implicit else condition is to continue..
   chop;
   @aclarr = &parse_current_line($_);

   $dongle = "";
   for ($i=0; $i<$nbits; $i++) {
      if ($outstr[$i]) {
         $dongle = $aclarr[$i].$dongle;
         }
      }

#########################################################################
# walk thru the hash table, incrementing bins that fits the key      #
#########################################################################

#cycle from 0 theu 255, and see what fits the dongle mask

for ($i = 0; $i < $endhash; $i++) {
    $bini = substr((unpack("B*", pack("n", $i))), -$outbits, $outbits);
    $fits_into_mask = 1;
    @aobini = split(//,$bini);
    @aodongle = split(//,$dongle);
    for ($j = 0; $j <$outbits; $j++) {
       if (($aobini[$j] ne $aodongle[$j]) && ($aodongle[$j] ne 'x')) {
          $fits_into_mask = 0;
          }
       }
    if ($fits_into_mask == 1) {
       $aoacls[$i] = $aoacls[$i] + $hwchecks;
       }
    }
}


#########################################################################
#     Print out the hash table, and each bin's total checks      #
#########################################################################

$max = 0; $min = 99999; $total = 0; $rows = ($endhash/8);
for ($i = 0; $i < ($rows-1); $i++) {
   $~ = 'REPORT2';
   if ($opt_v) {
      write;
      }
   for ($j = 0; $j < 8; $j++) {
      $max = ($aoacls[$i*8+$j] > $max) ? $aoacls[$i*8+$j] : $max;
      $min = ($aoacls[$i*8+$j] < $min) ? $aoacls[$i*8+$j] : $min;
      $total = $total + $aoacls[$i*8+$j];
```

```
        }
    }
$average = $total/$endhash;
print ("ACL2TREE ran on $filename\n");
print "Maximum checks: $max\nMinimum checks: $min\nAverage checks:$average\n";


#####################################################################
# Formats for various screen dumps                                  #
#####################################################################

format STDOUT_TOP =
  SourceAddr      DestAddr        Proto.   Port        StartPort  Key
  --------------  --------------  -------  -------     --------- --------
.

format STDOUT =
  @<<<<<<<<<<<<<  @<<<<<<<<<<<<<  @<<<<<<  @<<<<<<     @<<<<<<<  @<<<<<<<<<<
  $fullsa,        $fullda,        $protocol,$porttype, $startport,$dongle
.

format REPORT2 =
  @>>>:@<<< @>>>:@<<< @>>>:@<<< @>>>:@<<< @>>>:@<<< @>>>:@<<< @>>>:@<<< @>>>:@<<<
  ($i*8+0), $aoacls[$i*8+0],($i*8+1), $aoacls[$i*8+1],($i*8+2),
$aoacls[$i*8+2],($i*8+3), $aoacls[$i*8+3],($i*8+4), $aoacls[$i*8+4],($i*8+5),
$aoacls[$i*8+5],($i*8+6), $aoacls[$i*8+6],($i*8+7), $aoacls[$i*8+7]
.

#####################################################################
# Function definitions                                              #
#####################################################################


# function to calculate start port

sub calc_start_port {
   local($port) = @_;
   if (!$port) {
      return "any";
      }
   elsif (isdigit($port)) {
      return $port;
      }
   elsif ($holabels{$port}) {
      return $holabels{$port};
          }
   else {
      print "Error! Can't understand a port name of $port\n";
      die;
      }
   }


# A function to parse the current ACL entry, and return an array
# of values for SA, SA-MASK, DA, DA-MASK, PROTOCOL, DEST. PORT

sub parse_current_line{
   local($line) = @_;
   my @aclarr;
```

```
@word = split(" ",$line);
#getting protocol type
while (($w = shift(@word))
       && ($w !~ /udp/)
       && ($w !~ /tcp/)
       && ($w !~ /ip/)
       && ($w !~ /icmp/)
       && ($w !~ /igrp/)
       && ($w !~ /gre/)
       && ($w !~ /igmp/)
       && ($w !~ /nos/)
       && ($w !~ /ospf/)
       ) {
   };
$protocol = $hoprot{$w};

#getting source address
$w = shift(@word);
if ($w =~ /host/) {      #access-list 141 permit ip host 152.163.177.207
   $fullsa = shift(@word);
   $fullsamask = "0.0.0.0";
   }
elsif ($w =~ /any/) {    #access-list 141 permit tcp any
   $fullsa = $w;
   }
else {                   #access-list 141 permit tcp 198.81.7.0 0.0.0.255
   $fullsa = $w;
   $fullsamask = shift(@word);
   }
#getting destination address
$w = shift(@word);
if ($w =~ /host/) {      #access-list 141 permit ip host 152.163.177.207
   $fullda = shift(@word);
   $fulldamask = "0.0.0.0";
   }
elsif ($w =~ /any/) {    #access-list 141 permit tcp any
   $fullda = $w;
   }
else {                   #access-list 141 permit tcp 198.81.7.0 0.0.0.255
   $fullda = $w;
   $fulldamask = shift(@word);
   }
#getting port info
$porttype = "";
$startport = "any";
while ($w = shift(@word)){
   if ($w =~ /range/){
      $porttype = $w;
      $w = shift(@word);
      $startport = &calc_start_port($w);
      shift(@word);
      }
   elsif (($w =~ /gt/) | ($w =~ /eq/) | ($w =~ /lt/)) {
      $porttype = $w;
      $w = shift(@word);
      $startport = &calc_start_port($w);
      }
   elsif ($w =~ /established/){
      $porttype = $w;
```

```
      $startport = 1234;
      }
   else {
      $porttype = 'eq';
      $startport = &calc_start_port($w);
      }
   }

 ($sa[0],$sa[1],$sa[2],$sa[3]) = split(/\./, $fullsa);
 ($samask[0],$samask[1],$samask[2],$samask[3]) = split(/\./, $fullsamask);
 ($da[0],$da[1],$da[2],$da[3]) = split(/\./, $fullda);
 ($damask[0],$damask[1],$damask[2],$damask[3]) = split(/\./, $fulldamask);

if ($opt_v) {
   #not the best place to put this write, since $dongle hasn't been
   #define for the current ACL entry.
   write;
   }

#Convert these fields to binary, pushing them LSB first, into an array.

#Convert SA to binary. Use x's for any or for any bit that's masked
for ($j = 3; $j >= 0; $j--){
  @sabits = split(//,substr((unpack("B*", pack("n", $sa[$j]))), -8, 8));
  @smbits = split(//,substr((unpack("B*", pack("n", $samask[$j]))), -8, 8));
  for ($i=7; $i>=0; $i--) {
    $bit = ($smbits[$i] || ($fullsa =~ /any/))? 'x' : $sabits[$i];
    push(@aclarr,$bit);
    }
  }

#Convert dA to binary. Use x's for any or for any bit that's masked
for ($j = 3; $j >= 0; $j--){
  @dabits = split(//,substr((unpack("B*", pack("n", $da[$j]))), -8, 8));
  @dmbits = split(//,substr((unpack("B*", pack("n", $damask[$j]))), -8, 8));
  for ($i=7; $i>=0; $i--) {
    $bit = ($dmbits[$i] || ($fullda =~ /any/))? 'x' : $dabits[$i];
    push(@aclarr,$bit);
    }
  }

#Convert protocol to binary.
@protbits = split(//,substr((unpack("B*", pack("n", $protocol))), -8, 8));
for ($i=7; $i>=0; $i--) {
   $bit = $protbits[$i];
   push(@aclarr,$bit);
   }

#Convert port to binary.
@portbits = split(//,substr((unpack("B*", pack("n", $startport))), -16, 16));
for ($i=15; $i>=8; $i--) {
   $bit = ($porttype =~ /eq/) ? $portbits[$i] : 'x';
   push(@aclarr,$bit);
   }
for ($i=7; $i>=0; $i--) {
   $bit = $portbits[$i];
   push(@aclarr,$bit);
   }
```

```
    return @aclarr;
    }


# A function to convert from binary to decimal

sub bintodec {
    unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}
```

Experiment #1. Hashing on the 2nd octet of the destination address:

0:56	1:56	2:56	3:56	4:56	5:56	6:56	7:56
8:56	9:56	10:56	11:57	12:56	13:56	14:56	15:56
16:116	17:185	18:56	19:56	20:56	21:56	22:56	23:56
24:56	25:56	26:56	27:56	28:56	29:56	30:56	31:56
32:56	33:56	34:56	35:56	36:56	37:56	38:56	39:56
40:56	41:56	42:56	43:56	44:56	45:56	46:56	47:56
48:61	49:57	50:57	51:65	52:58	53:66	54:57	55:57
56:57	57:57	58:57	59:80	60:57	61:57	62:57	63:57
64:56	65:56	66:56	67:56	68:56	69:57	70:63	71:56
72:56	73:56	74:56	75:56	76:56	77:56	78:56	79:56
80:56	81:56	82:56	83:56	84:56	85:56	86:56	87:56
88:56	89:56	90:56	91:56	92:56	93:56	94:56	95:56
96:56	97:56	98:56	99:56	100:56	101:56	102:56	103:57
104:56	105:56	106:56	107:56	108:56	109:56	110:56	111:56
112:56	113:56	114:56	115:56	116:56	117:56	118:56	119:56
120:56	121:56	122:56	123:56	124:56	125:56	126:56	127:56
128:56	129:56	130:56	131:56	132:56	133:56	134:58	135:56
136:56	137:56	138:56	139:56	140:56	141:56	142:56	143:56
144:56	145:56	146:56	147:56	148:56	149:56	150:56	151:56
152:57	153:56	154:56	155:56	156:56	157:56	158:56	159:56
160:56	161:56	162:56	163:56	164:56	165:56	166:56	167:56
168:56	169:56	170:56	171:56	172:56	173:56	174:56	175:56
176:56	177:56	178:56	179:56	180:56	181:56	182:56	183:56
184:56	185:56	186:56	187:59	188:56	189:56	190:56	191:56
192:56	193:56	194:56	195:56	196:56	197:56	198:56	199:56
200:56	201:56	202:56	203:56	204:56	205:56	206:56	207:56
208:56	209:56	210:56	211:56	212:56	213:56	214:56	215:56
216:56	217:56	218:56	219:56	220:56	221:56	222:56	223:56
224:56	225:56	226:56	227:56	228:56	229:56	230:56	231:56
232:56	233:56	234:56	235:56	236:56	237:56	238:56	239:56
240:56	241:56	242:56	243:56	244:56	245:56	246:56	247:56
248:56	249:56	250:56	251:56	252:56	253:56	254:56	255:56

Maximum checks:185, Minimum checks:56

Experiment #2. Hashing on an 8bit combination of source address (4 bits of octet 2) and destination address (4 bits of octet 2):

0:12	1:119	2:11	3:30	4:16	5:11	6:11	7:11
8:11	9:11	10:11	11:14	12:11	13:11	14:11	15:11
16:29	17:138	18:29	19:65	20:34	21:29	22:29	23:29

24:29	25:29	26:29	27:32	28:29	29:29	30:29	31:29
32:11	33:119	34:11	35:30	36:16	37:11	38:11	39:11
40:11	41:11	42:11	43:14	44:11	45:11	46:11	47:11
48:34	49:221	50:34	51:61	52:40	53:34	54:34	55:34
56:36	57:34	58:34	59:37	60:34	61:34	62:34	63:34
64:11	65:119	66:11	67:30	68:17	69:11	70:11	71:11
72:11	73:11	74:11	75:14	76:11	77:11	78:11	79:11
80:11	81:119	82:11	83:30	84:16	85:11	86:11	87:11
88:11	89:11	90:11	91:14	92:11	93:11	94:11	95:11
96:11	97:119	98:11	99:30	100:17	101:11	102:12	103:11
104:11	105:11	106:11	107:14	108:11	109:11	110:11	111:11
112:11	113:119	114:11	115:30	116:16	117:11	118:11	119:11
120:11	121:11	122:11	123:14	124:11	125:11	126:11	127:11
128:12	129:120	130:12	131:33	132:17	133:12	134:12	135:12
136:12	137:12	138:12	139:15	140:12	141:12	142:12	143:12
144:11	145:119	146:11	147:30	148:16	149:11	150:11	151:11
152:11	153:12	154:11	155:14	156:11	157:11	158:11	159:11
160:11	161:119	162:11	163:30	164:16	165:11	166:11	167:11
168:11	169:11	170:11	171:14	172:11	173:11	174:11	175:11
176:14	177:122	178:14	179:33	180:19	181:14	182:14	183:14
184:14	185:14	186:14	187:17	188:14	189:14	190:14	191:14
192:11	193:119	194:11	195:30	196:16	197:11	198:11	199:11
200:11	201:11	202:11	203:14	204:11	205:11	206:11	207:11
208:11	209:119	210:11	211:30	212:16	213:11	214:11	215:11
216:11	217:11	218:11	219:14	220:11	221:11	222:11	223:11
224:11	225:119	226:11	227:30	228:16	229:11	230:11	231:11
232:11	233:11	234:11	235:14	236:11	237:11	238:11	239:11
240:11	241:119	242:11	243:30	244:16	245:11	246:11	247:11
248:11	249:11	250:11	251:14	252:11	253:11	254:11	255:11

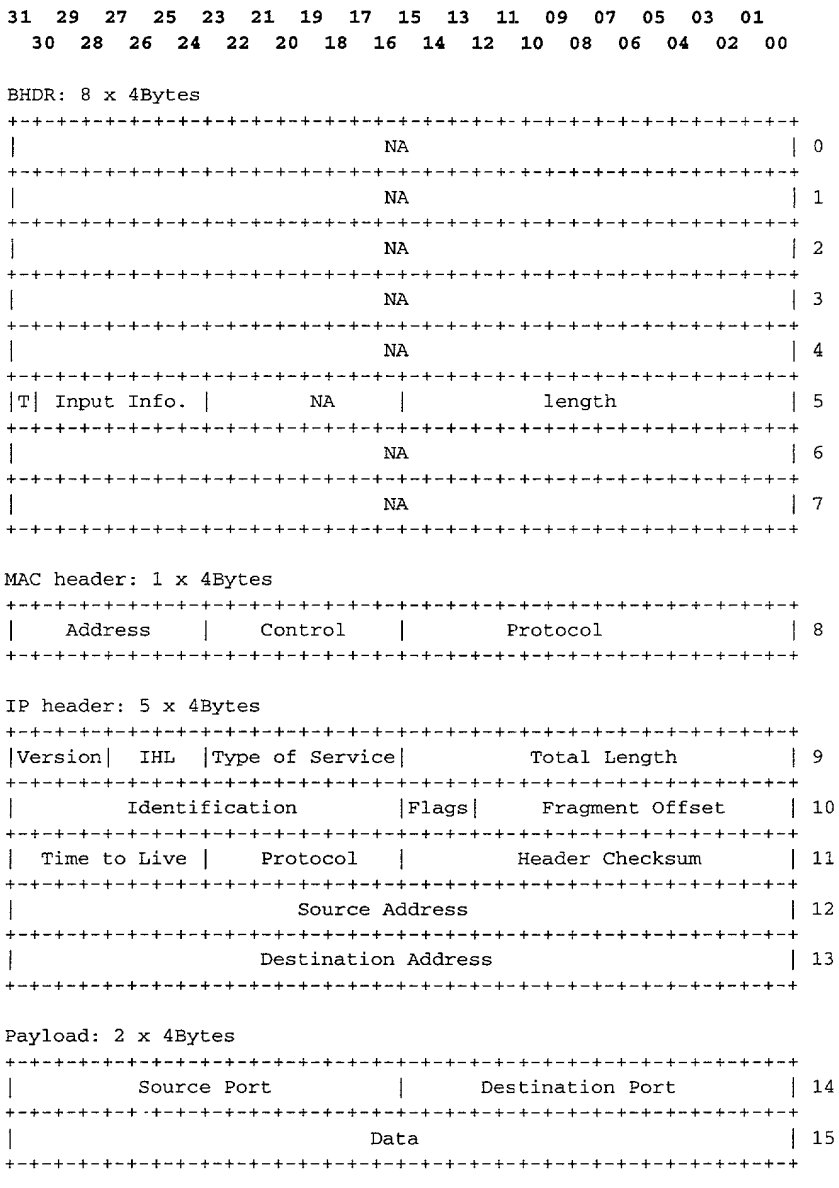
Maximum checks:221, Minimum checks:11

Experiment #3: Hashing on an 8 bit combination of destination address (4 bits of octet 2) and destination port (lower 4 bits):

0:4	1:54	2:4	3:23	4:7	5:4	6:4	7:4
8:5	9:4	10:4	11:4	12:4	13:4	14:4	15:4
16:4	17:61	18:4	19:9	20:6	21:4	22:4	23:4
24:5	25:4	26:4	27:4	28:4	29:4	30:4	31:4
32:6	33:56	34:6	35:11	36:8	37:6	38:6	39:6
40:7	41:6	42:6	43:6	44:6	45:6	46:6	47:6
48:3	49:60	50:2	51:7	52:4	53:2	54:3	55:2
56:3	57:3	58:2	59:2	60:2	61:2	62:2	63:2
64:2	65:51	66:2	67:7	68:4	69:2	70:2	71:2
72:3	73:2	74:2	75:2	76:2	77:2	78:2	79:2
80:22	81:67	82:22	83:27	84:25	85:22	86:22	87:22
88:23	89:22	90:22	91:24	92:22	93:22	94:22	95:22
96:2	97:59	98:2	99:11	100:4	101:2	102:2	103:2
104:4	105:2	106:2	107:2	108:2	109:2	110:2	111:2
112:7	113:64	114:7	115:12	116:10	117:7	118:7	119:7
120:8	121:7	122:7	123:7	124:7	125:7	126:7	127:7
128:2	129:51	130:2	131:7	132:4	133:2	134:2	135:2
136:3	137:2	138:2	139:2	140:2	141:2	142:2	143:2
144:3	145:46	146:3	147:8	148:6	149:3	150:3	151:3
152:4	153:3	154:3	155:3	156:3	157:3	158:3	159:3
160:16	161:59	162:16	163:22	164:19	165:16	166:16	167:16
168:17	169:16	170:16	171:16	172:16	173:16	174:16	175:16
176:2	177:51	178:2	179:10	180:4	181:2	182:2	183:2

```
184:3    185:2    186:2    187:2    188:2    189:2    190:2    191:2
192:2    193:44   194:2    195:7    196:4    197:2    198:2    199:2
200:3    201:2    202:2    203:2    204:2    205:2    206:2    207:2
208:2    209:51   210:2    211:11   212:4    213:2    214:2    215:2
216:3    217:2    218:2    219:2    220:2    221:2    222:2    223:2
224:2    225:58   226:2    227:7    228:4    229:2    230:2    231:2
232:3    233:2    234:2    235:2    236:2    237:2    238:2    239:2
240:8    241:72   242:8    243:28   244:11   245:8    246:8    247:8
248:9    249:8    250:8    251:9    252:8    253:8    254:8    255:8

Maximum checks:72, Minimum checks:2
```

Appendix B Prefetch Packet Header Format



Please note the following allowed values for each of the 5 bit decodes:

Table 5: A Summary of the selection choices for the ACL hash key

Selection Bits	KEY [7:6]	KEY [5:4]	KEY [3:2]	KEY [1:0]
00	SA[7:6]	SA[5:4]	SA[3:2]	SA[1:0]
01	SA[15:14]	SA[13:12]	SA[11:10]	SA[9:8]
02	SA[23:22]	SA[21:20]	SA[19:18]	SA[17:16]
03	SA[31:30]	SA[29:28]	SA[27:26]	SA[25:24]
04	DA[7:6]	DA[5:4]	DA[3:2]	DA[1:0]
05	DA[15:14]	DA[13:12]	DA[11:10]	DA[9:8]
06	DA[23:22]	DA[21:20]	DA[19:18]	DA[17:16]
07	DA[31:30]	DA[29:28]	DA[27:26]	DA[25:24]
08	SP[7:6]	SP[5:4]	SP[3:2]	SP[1:0]
09	SP[15:14]	SP[13:12]	SP[11:10]	SP[9:8]
0a	DP[7:6]	DP[5:4]	DP[3:2]	DP[1:0]
0b	DP[15:14]	DP[13:12]	DP[11:10]	DP[9:8]
0c	PROT[7:6]	PROT[5:4]	PROT[3:2]	PROT[1:0]
0d	TOS[7:6]	TOS[5:4]	TOS[3:2]	TOS[1:0]
0e	SA[3:2]	SA[1:0]	SA[7:6]	SA[5:4]
0f	SA[11:10]	SA[9:8]	SA[15:14]	SA[13:12]
10	SA[19:18]	SA[17:16]	SA[23:22]	SA[21:20]
11	SA[27:26]	SA[25:24]	SA[31:30]	SA[29:28]
12	DA[3:2]	DA[1:0]	DA[7:6]	DA[5:4]
13	DA[11:10]	DA[9:8]	DA[15:14]	DA[13:12]
14	DA[19:18]	DA[17:16]	DA[23:22]	DA[21:20]
15	DA[27:26]	DA[25:24]	DA[31:30]	DA[29:28]
16	SP[3:2]	SP[1:0]	SP[7:6]	SP[5:4]
17	SP[11:10]	SP[9:8]	SP[15:14]	SP[13:12]
18	DP[3:2]	DP[1:0]	DP[7:6]	DP[5:4]
19	DP[11:10]	DP[9:8]	DP[15:14]	DP[13:12]
1a	PROT[3:2]	PROT[1:0]	PROT[7:6]	PROT[5:4]
1b	TOS[3:2]	TOS[1:0]	TOS[7:6]	TOS[5:4]
1c	Reserved	Reserved	Reserved	Reserved
1d	Reserved	Reserved	Reserved	Reserved
1e	Reserved	Reserved	Reserved	Reserved
1f	Reserved	Reserved	Reserved	Reserved

Suggested values for this register are:

18_18_6_6 Key = {DP[3:2],DP[1:0],DA[19:18],DA[17:16]}

0e_18_6_6 Key = {SA[3:2],DP[1:0],DA[19:18],DA[17:16]}